# 9

# NOTE DI SOFTWARE

ARCHIVIO STORICO
ASSOCIAZIONE POZZO DI MIELE
FPDM-113
RVSW-107

UNIVERSITA'
DEGLI STUDI DI MILANO
ISTITUTO DI CIBERNETICA

Honeywell
Honeywell Information Systems Italia

**Note di software.**
è un bollettino trimestrale, edito a cura del Centro di Ricerca e Progettazione della Honeywell Information Systems Italia e dell'Istituto di Cibernetica dell'Università di Milano.

**Note di software.**
intende costituire uno strumento per la rapida e informale circolazione di informazioni, idee ed esperienze nell'area della tecnologia del software. In particolare, intende stimolare il dibattito sulle metodologie orientate alla diminuzione del rapporto costo/qualità dei programmi, e sugli strumenti atti ad automatizzare l'attività di produzione del software in ogni suo aspetto.

La collaborazione a **Note di software** è libera.
In particolare, si sollecitano contributi nella forma di brevi monografie o bibliografie essenziali commentate sui seguenti temi: tecniche e strumenti per la programmazione strutturata e modulare, metodologie e strumenti per il testing e il debugging dei programmi, documentazione del software, aspetti organizzativi e gestionali nella costruzione di sistemi software di grandi dimensioni.

Chi desiderasse pubblicare un contributo è pregato di prendere contatto con il Comitato di Redazione (HISI - Via ai Laboratori Olivetti - Pregnana Milanese), o di inviare direttamente il dattiloscritto in triplice copia.
I contributi non dovrebbero superare le quattromila parole.

I lavori accettati per la pubblicazione non vengono revisionati da un comitato tecnico. Pertanto, ogni contributo pubblicato riflette le opinioni personali dei suoi autori, e non necessariamente quelle del Comitato di Redazione.
Eventuali revisioni saranno effettuate di comune accordo fra il Comitato di Redazione e i proponenti.

**Note di software.**
viene distribuito a chi ne faccia richiesta alla redazione.

**Direttore responsabile:** Paolo Ghelfi

**Comitato di Redazione:** Roberto Polillo - Wanda Anselmetti

**Redazione:** Franco Soresini

# 9

FPDM-113

Gennaio - Marzo 1979

# NOTE DI SOFTWARE

## Indice:


**QUESTO NUMERO ....** pag. 1

**CONTRIBUTI TECNICI:**

— TESTGE: UNA TECNICA PER LA GENERAZIONE DI DATI DI PROVA AFFIDABILI ED ESAURIENTI PER APPLICAZIONI EDP, di C. Scuratti " 2

— LE RETI DI PETRI NELLA DESCRIZIONE DI PROCEDURE ORIENTATE ALLA PRODUZIONE DI SOFTWARE, di G. Degli Antoni, D. Marini, M. Parini, B. Zonta " 22

— BOOLEAN CONTROL OF PROGRAMMING: COME AUMENTARE IL GRADO DI AFFIDABILITA' DI PROCEDURE A SCARSA CONTROLLABILITA' INTERMEDIA, di G. Ciucci, A. Guarnieri, L. Quartapelle " 34

— APPROCCI ALLA COMUNICAZIONE DI ATTIVITA' CONCORRENTI IN SISTEMI DI ELABORAZIONE DISTRIBUITI, di G. Perego " 38

**AVVENIMENTI:**

— INFORMATION SYSTEMS, ORGANIZATIONAL CHOICE, SOCIAL VALUES, Pisa, 9-20 Aprile 1979, di F. De Cindio " 48

— 3° CONVENGNO SULLA PROGRAMMAZIONE STRUTTURATA, Milano, 23-25 Maggio 1979, di A. Cicu " 54

— TERZA CONFERENZA INTERNAZIONALE HONEYWELL SULL'INGEGNERIA DEL SOFTWARE, Bloomington, 27-29 Marzo 1979, di A. Cicu " 55

**IL SEGNALIBRO** " 59

# QUESTO NUMERO ...

*Le metodologie e gli strumenti per il testing dei programmi, e i linguaggi che permettono di descrivere, o progettare, sistemi di attività concorrenti, sono i temi a cui si rivolge questo nono numero di NOTE DI SOFTWARE.*

*Il primo lavoro dedicato al testing affronta il problema della progettazione di insiemi, affidabili e "completi", di dati di prova per applicazioni di tipo EDP. In esso, C. Scuratti presenta TESTGE, un "Test Generator" realizzato presso il Credito Italiano, e l'approccio da cui esso trae origine. La filosofia dello strumento è coerente con le metodologie di programmazione (quali quella di Warnier, di Jackson e PHOS) che guidano il programmatore nella "derivazione" del programma a partire da una descrizione, gerarchica, dei suoi dati di ingresso e di uscita. Così, TESTGE genera i flussi di ingresso per il programma in prova a partire da una descrizione logica della struttura complessiva dei suoi dati di ingresso, espressa in termini gerarchici, in una notazione linearizzata ma analoga a quella proposta dalle metodologie citate. In tal modo, a differenza di molti generatori di tests disponibili, che producono singoli files fra loro indipendenti, a partire da una descrizione di tipo fisico, TESTGE è in grado di gestire tutte le interrelazioni logiche fra dati presenti su files diversi, generando (eventualmente) tanti "giochi-prova" quanti sono necessari per coprire tutti i casi possibili.*

*Il testing di software matematico è, invece, l'ambito cui si rivolge la nota di G. Ciucci, A. Guarnieri e L. Quartapelle, in cui viene proposta una tecnica di controllo, al tempo di esecuzione, dei risultati intermedi del calcolo. L'adozione della tecnica viene proposta, ad esempio, per quei programmi per i quali sia noto, a priori, soltanto il "campo di credibilità" dei risultati del calcolo, ma non specifici valori attesi.*

*Il lavoro di G. Degli Antoni, D. Marini, M. Parini e B. Zonta propone l'uso delle reti di Petri, che qui vengono illustrate in modo informale, per rappresentare in modo rigoroso processi di produzione. Le reti di Petri, infatti, permettono di descrivere, in modo semplice e facendo uso di un numero molto limitato di concetti di base, la struttura statica e il comportamento dinamico di sistemi in cui più attività possano aver luogo concorrentemente, sulla base del verificarsi di specificate condizioni. In particolare, l'articolo propone, presentandone un esempio, l'uso di tale strumento come supporto alla gestione di processi di produzione di software, accennando alle relazioni con tecniche PERT e alla possibilità di "tradurre" le reti in frasi non ambigue del linguaggio naturale.*

*Infine, il lavoro di G. Perego esamina e pone a confronto due recenti proposte di costrutti ad alto livello per linguaggi orientati alla programmazione concorrente. Le proposte, quella dei "Processi Distribuiti" di Brinch Hansen e quella dei "Processi Sequenziali Comunicanti" di Hoare, tentano di superare alcuni limiti del concetto di "monitor", presentato dallo stesso Hoare in anni recenti e adottato nel linguaggio di programmazione Concurrent Pascal, ad esempio nell'applicazione a sistemi distribuiti senza memoria comune.*

*La Redazione*

# CONTRIBUTI TECNICI

## TESTGE: UNA TECNICA PER LA GENERAZIONE DI DATI DI PROVA AFFIDABILI ED ESAURIENTI PER APPLICAZIONI EDP

C. Scuratti
Credito Italiano, Milano e Univ. di Milano, Istituto di Cibernetica

**Riassunto**

Questo articolo descrive TESTGE, una tecnica e uno strumento nuovi per la generazione automatica di dati di test affidabili per un programma EDP.
L'idea di base consiste nel fornire un'unica descrizione logica, top-down, dell'insieme globale dei dati di input al programma da collaudare, in modo da consentire al generatore la gestione automatica di tutte le connessioni possibili fra i dati.
Lo strumento realizzato è in grado di fornire ottimi dati di test, ed una serie di informazioni utili sia in fase di collaudo che a fini documentativi.
L'articolo si divide in due parti. Nella prima, constatata la necessità, e la difficoltà, di predisporre dati di test efficienti e l'inadeguatezza degli attuali "Test Data Generators", si illustra la nuova tecnica, mutuata da esperienze di programmazione strutturata, e le caratteristiche generali dello strumento realizzato. La seconda parte approfondisce alcune caratteristiche dello strumento, con l'aiuto di un esempio completo di uso di TESTGE.

**Abstract**

This paper describes TESTGE, a new technique and a new automatic tool to generate reliable test data for EDP programs. The basic idea is the use of a unique logical top-down description of the input data of the program to be tested, so that the generator can automatically manage all expected relationships between the data. The implemented tool supplies excellent test data, and useful information for testing and program documentation.
The first part of the paper discusses the need, and the difficulties, of designing efficient test data, the inadequacy of current "Test Data Generators", the new technique, based on experiences of structured programming, and the general features of the implemented tool.
In the second part, the generator is illustrated, with the example of a complete TESTGE application.

### INTRODUZIONE

Il presente articolo propone un nuovo approccio "scientifico" e sistematico alla fase di testing di un programma, e un opportuno strumento di supporto: TESTGE (Test Generator).

Constatata l'inadeguatezza dei vari "test generators" attualmente in circolazione (generalmente più che di test generators si tratta di "files generators") nonchè l'impraticabilità al momento di varie proposte teoriche (verifying compiler, linguaggi interattivi di testing ecc.), basandosi sull'approccio allo studio della struttura dei dati introdotto dalle diverse metodologie di costruzione programmi e principalmente sulla proposta di Warnier, si cerca di fornire all'utente un modo chiaro e scientifico di stesura dei test nonchè uno strumento che l'aiuti nell'espletamento di tale funzione.

Nella prima parte dell'articolo lo strumento TESTGE viene descritto nelle sue linee essenziali, mettendone in risalto la filosofia sottostante. Nella seconda parte, ne viene dato un esempio di applicazione completo, anche se non esauriente di tutte le possibilità dello strumento.

Nella nota vengono sottolineati i vantaggi attesi da un utilizzo sistematico dello strumento, quali ad esempio:

- risparmio di tempo nella stesura dei tests
- la certezza di avere dei tests di buon livello
- l'introduzione di un linguaggio per la descrizione dei dati
- la possibilità di utilizzare tale linguaggio nella documentazione del programma
- il facile recupero dei tests in fase di manutenzione
- possibilità di un controllo della bontà dei tests da parte del responsabile dell'applicazione.

Si accenna inoltre ai possibili sviluppi futuri di TESTGE.

## PARTE I: L'APPROCCIO DI TESTGE

### 1. TESTING DI UN PROGRAMMA

L'argomento che qui vogliamo trattare riguarda quella fase del lavoro di un programmatore che va sotto il nome generico di prova dei programmi. È indubbio, infatti, che il testing di un programma e il relativo debugging rappresentano una buona percentuale del tempo dedicato dal programmatore alla soluzione di un problema e per di più, secondo la nostra esperienza, non è tra le attività predilette dal programmatore stesso. Tuttavia, se è vero, come è vero, che il programma esatto è un'astrazione teorica, nondimeno esso è la meta cui ognuno tende nella realizzazione di un programma. Ciò nonostante ci sembra che, alle innovazioni ultimamente intervenute soprattutto nella fase di disegno del programma con l'introduzione delle tecniche di programmazione strutturata, non è corrisposto un analogo sviluppo nelle tecniche di testing e debugging.

Prima di procedere, comunque, è opportuno delimitare il campo del discorso e chiarire il significato di alcune espressioni che utilizzeremo nel seguito.

Innanzitutto, il discorso è per ora limitato alla produzione di software applicativo così come viene normalmente realizzata in ambienti edp tradizionali. Il termine software applicativo è, in effetti, abbastanza ampio nel suo significato e nel suo contenuto, tuttavia riteniamo che sia sufficientemente chiaro come concetto e che si chiarirà meglio nel prosieguo del discorso.

L'azienda agisce inserita in una certa realtà che si evolve sia in base a fattori esterni all'azienda sia in base a fattori indotti dall'azienda stessa.

I dati non sono, in definitiva, che una rappresentazione il più fedele possibile del mondo esterno e le trasformazioni di tale realtà vengono simulate tramite programmi che agiscono su dati che la rappresentano.

In genere tali programmi sono riuniti per aree omogenee di applicazione e costituiscono le cosiddette procedure edp; avremo quindi, ad esempio, la procedura paghe e stipendi, la procedure fatturazione ed IVA ecc.

È ovvio a questo punto come, essendo i dati una rappresentazione della realtà, il lavoro di scelta di tali dati sia fondamentale nell'ambito della realizzazione di una procedura; è ovvio anche che tale rappresentazione, come qualsiasi modello, ha in sè un certo grado di approssimazione: è compito della persona addetta alla realizzazione della procedura determinare il grado di approssimazione accettabile.

Abbiamo quindi individuato nell'ambito della produzione del software applicativo una suddivisione per procedure e nell'ambito della procedura una suddivisione per programmi.

Ad una tale suddivisione gerarchica del software edp corrisponde in genere un'analoga suddivisione gerarchica del personale addetto alla produzione del software: analisti, che si interessano della procedura in generale, della suddivisione della procedura in programmi e della descrizione in maniera più o meno dettagliata delle trasformazioni che ciascun programma deve effettuare sui dati di input per ottenere i dati di output, nonchè ovviamente della definizione particolareggiata dei dati coinvolti, e programmatori, che traducono le indicazioni fornite dall'analista in un linguaggio comprensibile all'elaboratore controllando nel contempo che il lavoro svolto concordi con quanto richiesto.

Non entriamo qui in maggiori dettagli per quanto riguarda il lavoro di analisi ed esaminiamo invece un po' più a fondo il lavoro svolto dal programmatore nella realizzazione di un programma.

Riassumendo, possiamo definire un programma come una trasformazione più o meno complessa di un insieme di dati in ingresso in un ben definito insieme di dati in uscita, trasformazione che realizza certe funzionalità riscontrabili nel mondo reale.

Ora nella realizzazione di un programma possiamo distinguere a grandi linee tre passi principali e precisamente:

- studio delle specifiche di programmazione e analisi
- disegno e codifica
- testing e debugging.

In questo scritto ci occuperemo quasi esclusivamente della terza fase con qualche accenno alle due precedenti.

Innanzitutto in tale fase, genericamente detta collaudo del programma, è opportuno tenere distinte le due attività di testing e debugging, benchè generalmente i due termini vengano utilizzati in modo

intercambiabile. L'attività di testing consiste nel determinare l'esistenza o meno di errori nel programma, l'attività di debugging localizza la causa dell'errore e ne esegue la correzione. È chiaro quindi come le due attività risultino concettualmente diverse anche se molto legate tra di loro.

"Testare" un programma significa, quindi, verificare che il prodotto programma realizzato si comporti in maniera corretta per tutti i possibili insiemi di dati in input, dove "comportarsi in maniera corretta" significa fornire in uscita i risultati previsti.

Ciò implica, da parte del programmatore, la progettazione e la predisposizione di un campione di insiemi di dati di input sufficientemente rappresentativo di tutti gli insiemi di dati possibili e la verifica del comportamento corretto del programma. È ovvio, infatti, come il programmatore non possa riuscire ad esaminare il comportamento del programma per tutti i possibili insiemi di dati in input in quanto ciò, oltre ad essere in alcuni casi materialmente impossibile, comporterebbe comunque un notevole dispendio di energie e di tempo.

La scelta di tale campione viene, in genere, effettuata esaminando le specifiche di analisi e resta comunque una scelta soggettiva del programmatore con, in definitiva, una notevole possibilità di errore e soprattutto di incompletezza.

Resta infine da sottolineare che ben poche indicazioni sono a disposizione del programmatore su come costruire il suo campione di dati ed inoltre ogni programmatore, essendo in buona fede convinto di aver costruito un buon programma, ha la tendenza a ritenere esaurita la sua fase di collaudo prima di aver esaminato un campione di dati sufficientemente esaustivo.

## 2. SITUAZIONE ATTUALE

Attualmente, nonostante i diversi studi che si vanno facendo sull'argomento, gli unici aiuti a disposizione del programmatore sono puramente strumentali e sono dati dai diversi "test data generators" attualmente in commercio.

In genere tali programmi hanno in ingresso una descrizione fisica del flusso e di ogni singolo record del flusso e schede dati per ogni record da generare e forniscono in uscita un flusso generato secondo la struttura fisica richiesta.

Tali oggetti possono essere più o meno sofisticati per risparmiare al programmatore del lavoro manuale noioso, resta comunque il fatto che in genere non daènno al programmatore alcuna indicazione utile su come costruire dati di test efficienti. In genere il lavoro del programmatore consiste nel riprendere in mano le specifiche di programmazione e cercare di dedurre da esse quali possano essere le diverse occorrenze dei dati in ingresso in relazione ovviamente alle elaborazioni richieste dalle specifiche stesse. Ciò porta necessariamente ad alcuni inconvenienti che possono essere:

. incompletezza dei test generati in quanto alcune situazioni possibili non vengono previste

. errori materiali nel predisporrele chiavi di accoppiamento tra i diversi flussi, per cui alcune situazioni che si considerano previste non vengono in effetti esaminate

. proliferazione dei tests di prova in quanto, in generale, non basta un solo campione di tests per esaminare tutti i casi previsti dal programma. Ciò porta in genere a perdere il controllo dei tests già effettuati nonchè a seguire con difficoltà situazioni di errore generate nel programma da correzioni successive, in quanto i tests precedentemente utilizzati diventano difficilmente recuperabili.

Ci sono, in verità, diversi studi e diversi tentativi per cercare di evitare un tale dispendio di energie e di tempo e per cercare di rendere più scientifica la fase di testing. Uno di questi è il tentativo di creare dei "verifying compilers", dei compilatori cioè che, oltre a generare il programma compilato, si incaricano di verificare che quanto richesto venga eseguito correttamente. Ciò si otterrebbe inserendo nel testo del programma delle proposizioni specificanti le relazioni tra le diverse variabili del programma stesso, di modo che il compilatore possa automaticamente controllare che le proposizioni così inserite siano in accordo con quanto effettuato dal programma e segnalare, in caso di disaccordo, la ragione dell'errore. Ciò ovviamente renderebbe pressochè inutile il debugging del programma e darebbe al programmatore al certezza di aver costruito un programma corretto o per lo meno consistente con le asserzioni sulle variabili inserite nel programma. Sono ovvie le difficoltà di un tale tipo di approccio e precisamente:

al momento è praticamente impossibile costruire un compilatore del genere che riesca ad interpretare delle asserzioni scritte in linguaggio quasi naturale

. in un programma che non sia puramente scientifico e di calcolo è difficile individuare quali sono le variabili interessanti e tenere sotto controllo gli effetti collaterali

. il programmatore deve comunque controllare che le proposizioni inserite siano in accordo con le specifiche di programmazione, in quanto tali proposizioni possono essere consistenti col programma ma non risolvere il problema.

Altri autori propongono, come metodo di prova, la copertura topologica del programma da parte dei tests e a questo proposito esistono degli strumenti atti a controllare e a suggerire gli eventuali ulteriori tests da predisporre per la copertura dei rami di programma rimasti scoperti. In questo caso, però, oltre al fatto che il programmatore deve comunque predisporre i nuovi tests pur avendo qualche indicazione supplementare su come costruirli, ci sembra che il difetto principale consista nel fatto che la preparazione dei tests è del tutto slegata dalle specifiche di analisi, e quindi rimanga il pericolo di costruire un programma che esegue correttamente le sue operazioni ma non esaurisca tutte le richieste specificate dall'analisi.

Altri tentativi sono rivolti verso linguaggi di debugging che agiscono in ambiente interattivo e verso altri strumenti più o meno sofisticati, ma, per concludere, si può dire che gli strumenti a disposizione del programmatore sono abbastanza poveri e che alcune proposte, seppure molto avanzate dal punto di vista teorico, sono al momento impraticabili per un utilizzo esteso in ambienti edp tradizionali.

## 3. PROGRAMMAZIONE STRUTTURATA E TESTING TOP—DOWN

L'introduzione della programmazione strutturata nella produzione del software applicativo ha inciso notevolmente sul miglioramento della fase di disegno del programma, fornendo al programmatore degli ottimi strumenti concettuali e delle regole precise di comportamento, ma ha senz'altro inciso anche sulla fase di testing o, meglio, di debugging, per le ragioni che esporremo nel seguito.

Col termine programmazione strutturata vogliamo qui indicare le moderne tecniche di disegno e costruzione dei programmi, che si basano su alcuni concetti fondamentali ormai ampiamente accettati:

- sviluppo top-down della struttura del programma dedotto dalla struttura dei dati

- utilizzo, nella stesura, delle tre strutture fondamentali, tutte ad un ingresso ed una uscita: sequenza, selezione, iterazione, che permettono di avere un buon programma dal punto di vista della leggibilità.

In base a tali concetti vari autori hanno sviluppato e proposto diverse tecniche di disegno del programma, tra le più note citiamo la metodologia Warnier, la metodologia Jackson e infine la metodologia Phos.

Non entriamo qui nel merito delle singole metodologie, in quanto già se ne è ampiamente discusso nella letteratura, ma esaminiamo solo quali possono essere i riflessi dell'utilizzo delle tecniche citate sulla fase di testing e debugging del programma.

Diciamo subito che uno degli scopi principali della programmazione strutturata è quello di avere dei programmi pù corretti già in fase di disegno e costruzione, secondo il principio che per avere dei programmi esenti da errori è bene non mettercene fin dall'inizio e non, invece, tentare di correggerli in fase di prova. Anche perchè, come dice Dijkstra, la fase di testing "può provare la presenza di errori ma non la loro assenza".

Ora se tale scopo è raggiunto, è ovvio che ciè ridurrà di molto il lavoro del programmatore in fase di collaudo, molto di più, comunque, per quanto riguarda il debugging che non per quanto riguarda il testing poichè, pur con programmi a priori più corretti, il lavoro di predisposizione di dati di test rimarrà sostanzialmente lo stesso.

Un altro suggerimento ci viene ancora dalla programmazione strutturata, ed è quello di sfruttare la tecnica top-down seguita nella costruzione del programma anche per il testing del programma stesso.

In effetti, seguendo la tecnica di costruzione del programma in modo top down, noi avremo a disposizione sempre un programma completo a livello più basso in cui il programma diventa completamente eseguibile. È ovvio che ai livelli più alti avremo specificate solo le funzionalità principali del programma. Tali funzionalità saranno via via sottospecificate fino a raggiungere il dettaglio più basso coincidente col codice del linguaggio in cui il programma è scritto.

Niente comunque ci impedisce, già ai livelli più alti, di provare il nostro programma ricorrendo ad opportuni accorgimenti, mascherando i segmenti non ancora specificati con segmenti vuoti o inserendo "display" indicanti l'avvenuta esecuzione del segmento.

Ciò porta due vantaggi fondamentali:

1) i dati di test da preparare risultano molto semplici in quanto il codice da esaminare risulta sempre ridotto

2) la logica principale del programma, essendo anche logica a più alto livello, è la prima ad essere specificata ed è quindi quella che viene più volte provata.

Secondo noi, comunque, è necessario studiare a fondo il problema prima di procedere ad un testing top-down e avere ben chiaro quali sono le funzionalità da specificare per prime, per evitare una duplicazione di sforzi sia nella stesura di moduli fittizi che nella predisposizione di dati di test.

Concludendo quindi questa prima parte riguardante il testing in ambienti di programmazione strutturata, possiamo dire che:

- la fase di debugging viene molto ridotta in quanto la logica di costruzione del programma permette di evitare buona parte degli errori, ed inoltre la stessa logica permette di correggere gli errori rimanenti in maniera più semplice e più sicura;

- la fase di testing viene ridotta nella sua complessità adottando tecniche di testing top-down, purchè si scelga a priori un piano di azione che permetta di recuperare i dati di tests generati per i livelli superiori senza complicare la stesura di moduli fittizi.

Vedremo nel paragrafo successivo come ulteriori vantaggi si possano trarre dallo studio della struttura dei dati, passo fondamentale per l'applicazione delle tecniche di programmazione strutturata.

## 4. LO STUDIO DELLA STRUTTURA DEI DATI

Lo studio della struttura dei dati è senz'altro uno dei punti fondamentali e più innovativi delle moderne tecniche di costruzione dei programmi ed è su di esso che ruota, in effetti, tutto il lavoro di disegno del programma. Ciò porta, oltre che a costruire un programma più corretto, ad avere anche, effetto da non trascurare, un programma più facilmente comprensibile sia al programmatore stesso in tempi successivi che a terze persone che vi dovessero intervenire, in quanto a priori già a conoscenza che il programma stesso rifletterà in qualche modo la struttura dei dati coinvolti.

(Tuttavia, per inciso, si può notare che all'accettazione teorica dell'importanza dello studio dei dati nel disegno del programma non corrisponde un analogo comportamento da parte del programmatore nell'applicazione pratica. Infatti lo studio dei dati viene in genere ritenuto esaurito senza un adeguato approfondimento e nella correzione di un errore scoperto in fase di testing non si riparte, in genere, dalla struttura dei dati bensì unicamente dalla procedura. È chiaro come tutto ciò derivi da motivazioni pratiche, dovute soprattutto alla mancanza di un linguaggio per la descrizione dei dati che non sia puramente grafico e alla mancanza di una analoga fase di testing sui dati come sulla procedura. È evidente anche come un tal modo di procedere, su strutture di dati non corrette o perlomeno incomplete, possa portare ad errori non facilmente recuperabili, nonchè a programmi difficilmente comprensibili e quindi poco manutenibili).

A questo punto però è opportuno fare un breve cenno sui diversi approcci delle tre metodologie citate (Warnier, Jackson e Phos) allo studio dei dati. Ciò ci sarà di aiuto in seguito quando si parlerà del flusso di input a TESTGE.

Tutte e tre le tecniche esaminano la logica gerarchica dei dati utilizzando le tre strutture di base: sequenza, selezione, iterazione. La differenza fondamentale tra Jackson, Warnier e Phos sta nel privilegiare l'aspetto logico dei singoli flussi fisici distinti tra di loro, ovvero l'aspetto logico globale dell'insieme dei dati di input come unico flusso in ingresso e analogamente per i dati di uscita.

Warnier suggerisce di guardare l'insieme di tutti i dati di input come flusso unico e di descrivere quindi la struttura logica mettendo in risalto le varie interrelazioni significative per il problema.
Analoga operazione per l'output.

Jackson, invece, propone la descrizione logica separata di ciascun flusso fisico in input o in output.

La metodologia Phos, infine, si situa a metà strada tra le due, in quanto accetta un'impostazione analoga e Warnier per quanto riguarda lo studio dei

dati di output, mentre si rifà praticamente a Jackson per la descrizione dell'input.

Esaminate così sommariamente le differenze tra le tre metodologie per quanto riguarda l'approccio allo studio dei dati, a noi preme soprattutto sottolineare l'importanza del concetto di flusso logico unico di input e di output per la deduzione di un programma corretto. In effetti, lo strumento TESTGE, che verrà introdotto nel prossimo paragrafo, si basa, nella sostanza, su questo approccio.

D'altra parte, anche Jackson, pur non parlando esplicitamente di flusso logico unico, suggerisce, prima di passare al disegno della struttura del programma, di esaminare le connessioni logiche dei flussi ai vari livelli mettendone in risalto le diverse occorrenze. È chiaro infatti che è dall'intercalarsi tra loro dei flussi in ingresso e in uscita che nasce la complessità del programma.

Secondo noi è essenziale, prima di procedere al disegno del programma, avere ben dettagliata la struttura logica dell'insieme dei dati di input nonchè dei dati di output e a volte (quasi sempre) può essere necessario passare per la struttura dei singoli flussi fisici. Riassumendo, possiamo dire che per costruire un programma corretto sono necessarie le seguenti elaborazioni sulla struttura dei dati:

. costruzione della struttura dei singoli flussi fisici di input e output
. controllo delle singole strutture
. deduzione, dalle strutture predette, della struttura unica dei dati di input e della struttura unica dei dati di output mettendo in risalto le connessioni logiche rilevanti per il problema che si affronta
. controllo della struttura logica di input e della struttura logica di output
. derivazione del programma

Certo, la procedura può sembrare ridondante ma, secondo la nostra esperienza, questo è l'unico modo per arrivare ad un programma corretto senza dover rincorrere a posteriori situazioni non previste.

È chiaro, per tornare al testing dei programmi, che lo studio tanto approfondito dei dati non potrà avere che effetti benefici nella preparazione dei tests in quanto il programmatore sarà avvantaggiato nel suo lvoro dalla perfetta conoscenza dei dati acquisita in questa fase.

## 5. CARATTERISTICHE DI TESTGE

Tenendo presente tutte le indicazioni di cui sopra è stato pensato e realizzato TESTGE.

Descriviamo dapprima quali siano le caratteristiche di TESTGE, rimandando alla fine alcune considerazioni sugli scopi di tale realizzazione e sui vantaggi che riteniamo se ne possano trarre.

TESTGE è un generatore di veri e propri tests per un programma, che si differenzia da tutti gli altri pseudo-generatori che in realtà sono solo generatori di files. In effetti mentre gli altri generatori noti producono un file per volta lasciando al programmatore l'incombenza di attribuire a files diversi opportuni codici per evidenziarne le interrelazioni logiche, TESTGE gestisce automaticamente tutte le interrelazioni producendo una serie di dati di prova atti a ricoprire tutte le connessioni logiche previste, generando, se del caso, tanti giochi prova quanti sono necessari per coprire tutti i casi possibili.

I dati di input richiesti per la generazione dei dati di test sono di tre tipi:

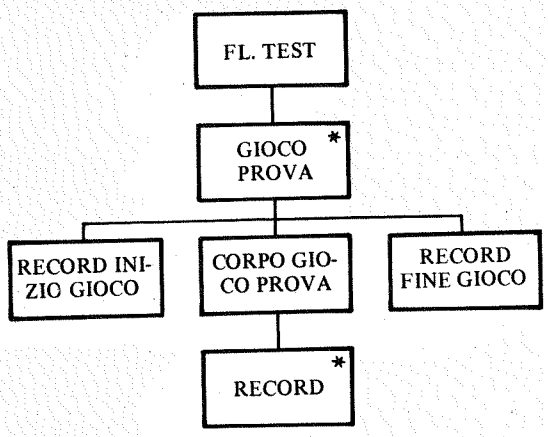
a) una descrizione della struttura gerarchica dell'insieme dei dati di input al programma per cui si vogliono generare dei tests
b) una descrizione di ciascun flusso fisico di input del programma
c) una descrizione di chiavi, tabelle, costanti, ranges coinvolti nella generazione dei tests.

TESTGE agisce su tali dati nel seguente modo. Dopo opportuni controlli sulla validità della struttura e sulla bontà dei singoli dati, nonchè sulla loro congruenza, TESTGE percorre tutta la struttura, generando per ogni ramó i records specificati sui diversi flussi. È ovvio che in tale percorso TESTGE terrà conto di tutte le particolarità specificate, cioè, in particolare, iterazioni e selezioni.

La descrizione su riportata é alquanto succinta ma crediamo che renda sufficientemente l'idea dei concetti che sono alla base di TESTGE. Nel paragrafo successivo sarà data una descrizione più dettagliata del linguaggio utilizzato per la definizione dei dati di input; qui di seguito invece diamo un cenno sugli outputs da TESTGE.

Abbiamo innanzitutto l'output principale, costituito dai singoli flussi di tests per il programma da collaudare, e alcuni outputs di supporto riguardan-

ti in genere segnalazioni di errori e incompatibilità e,soprattutto, l'esposizione in chiaro della struttura dati introdotta con relativi diagnostici e riferimenti. Per quanto riguarda gli outputs di supporto, è sufficiente fare riferimento all'esempio commentato che segue. Parliamo invece un pò più diffusamente dei singoli flussi di tests. Innanzitutto è da chiarire che tali flussi saranno tanti quanti sono i flussi in input al programma da collaudare e che ogni flusso conterrà in genere più giochi prove (in numero uguale per ciascun flusso) secondo la struttura seguente.




(La grafica utilizzata è quella di Jackson con il simbolo * ad indicare ripetizione e il simbolo ° ad indicare alternativa).

Come si vede, oltre ai veri e propri records di tests,abbiamo dei records spuri che funzionano da separatori, uno all'inizio e uno alla fine di ciascun gioco prova. È ovvio che prima di utilizzare tali flussi per la prova del programma interessato bisognerà ripulirli dei records spuri, cosa abbastanza agevole da farsi con un programma di utilità in genere disponibile su ogni elaboratore. Per agevolare tale compito, ogni generazione sarà accompagnata da un report indicante il numero di giochi prova generati e, distintamente per ogni flusso, la posizione dei records di inizio e fine gioco prova. (È ovvio che qualora per un gioco prova un flusso dovesse mancare, ad es. per flussi opzionali, per quel flusso e per quel gioco prova verranno generati solo i records di inizio e fine gioco prova).

## 6. DATI IN INPUT A TESTGE

Abbiamo accennato, nel paragrafo precedente, ai diversi tipi di dati richiesti da TESTGE per la generazione di dati di tests. Ne diamo qui una descrizione più dettagliata, partendo dalla struttura dei dati che è senz'altro la parte più difficile e nello stesso tempo la più importante.

In prima approssimazione, la tecnica utilizzata per la descrizione dei dati è la tecnica suggerita da Warnier e cioè quella di descrivere tutti i dati di input come appartenenti ad un unico flusso di input,mettendo in rilievo le connessioni logiche tra dati provenienti da flussi diversi al livello cui si verificano. Abbiamo detto in prima approssimazione in quanto, a differenza di Warnier, il flusso di input a TESTGE deve contenere solo alternative esclusive e non anche alternative ad "or inclusivo".

Prima di passare alla descrizione del linguaggio vero e proprio è opportuno fare un cenno sull'organizzazione generale della struttura.
Come già detto, i dati vengono studiati in modo top-down livello per livello, sottospecificando via via i livelli fino ad arrivare al livello elementare intendendo come tale il livello di singolo record.
Tale procedimento viene riportata anche nella struttura per TESTGE che viene divisa in sottostrutture gerarchiche richiamate dalle strutture di livello superiore tramite CALL.

Ogni struttura è identificata da un nome sulla scheda STRUCTURE ed è chiusa da una ENDSTR, cioè:

```
«nome-struttura»      STRUCTURE

                      ENDSTR
```

La CALL ovviamente farà riferimento al «nome-struttura» della scheda STRUCTURE, ad es.:

```
CALL «nome-struttura»
```

È ovvio che una struttura non può richiamare se stessa nemmeno attraverso un struttura da essa richiamata (come CALL e PERFORM del linguaggio COBOL) ed inoltre la struttura principale, che deve essere sempre la prima, non sarà mai richiamata.

La descrizione dei dati nelle singole strutture utilizza le tre strutture classiche della programmazione strutturata:

. sequenza
. selezione
. iterazione

La sequenza non ha alcuna rappresentazione nel linguaggio; per un insieme di dati alternativo, abbiamo invece la rappresentazione:

```
IF      «condizione»
ELSE
ENDIF
```

dove «condizione» è una sequenza di caratteri qualsiasi, che ha esclusivamente funzioni di commento. (Attualmente è prevista solo l'alternativa a due vie come quella rappresentata. In effetti, si tratta di una limitazione solo apparente in quanto nulla vieta di porre altre alternative nel ramo dell'IF o dell'ELSE e così via. Cioè ad esempio:

```
IF      «condizione»
ELSE
    IF      «condizione»
    ELSE
        IF
        ENDIF
    ENDIF
ENDIF
```

che, in effetti, è un'alternativa a più vie in cui l'unica complicazione è data dal proliferare di ENDIF e dall'avere le parole chiavi ELSE ed IF su due schede distinte).

Per un insieme di dati ripetitivo abbiamo invece

```
ITER    Ki, Kj,...
REPEAT
```

in cui Ki, Kj ecc. indicano le chiavi interessate alla iterazione.

Oltre alle due strutture su riportate è prevista l'istruzione esecutiva

```
DO XXX YYY
```

che richiede la generazione del record di nome XXX, per il flusso YYY.

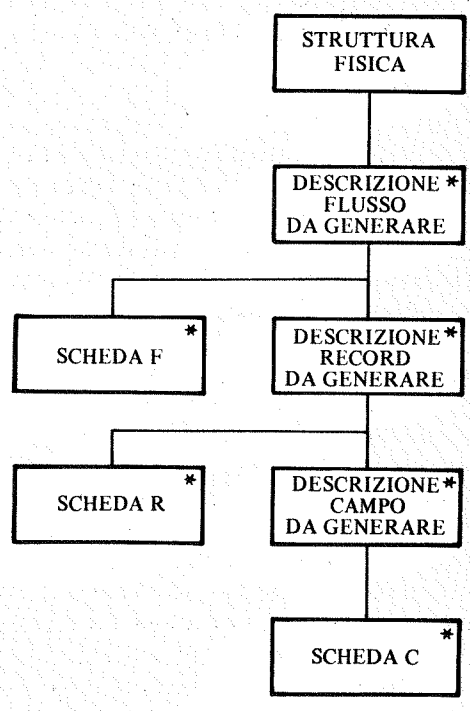
Esempio (vedi Parte II) :

```
FIL-SOLO-ARK    STRUCTURE
                DO 010 ARK
                ITER K1
                    DO 020 ARK
                REPEAT
                DO 019 ARK
                ENDSTR
```

In questo caso, la struttura logica FIL—SOLO—ARK richiede la generazione, per il file ARK, di un record tipo 010 (definito a parte), seguito da un certo numero di record di tipo 020 (definiti a parte), seguiti da un record di tipo 019 (definito a parte). La chiave (variabile da record a record generato, secondo una legge definita a parte) dei record 020 ha nome K1.

Questo è tutto per quanto riguarda la descrizione della struttura dei dati. Oltre alla struttura logica, come abbiamo già detto, è prevista una descrizione fisica di ogni singolo flusso e dei record del flusso.

La struttura di questa descrizione fisica è la seguente:




Come si vede, per ogni flusso sono previste una o più schede di tipo F contenenti l'elenco dei tipi record del flusso. Quindi, per ogni tipo di record sono previste una o piú schede R specificanti i singoli

campi del record in oggetto. Per qualche campo del record può essere necessaria un'ulteriore descrizione (scheda C) specificante per quel campo il contenuto e/o eventuali operazioni da farsi. I contenuti dei singoli campi possono essere ricavati da tabelle o da costanti o da memorizzatori o da totalizzatori, e le operazioni che si possono fare su tali campi sono in genere operazioni di memorizzazione, totalizzazione o azzeramento.

Le tabelle e le costanti di cui sopra nonchè le chiavi per le iterazioni vengono descritte nella terza parte dei dati di input, rispettivamente con schede di tipo T, L e K.

Si rimanda alla seconda parte per maggiori dettagli.

## 7. CONSIDERAZIONI

TESTGE è utile per i seguenti principali motivi:

a) abbrevia il tempo necessario per la predisposizione di tests

b) evita al programmatore lo studio delle diverse combinazioni possibili tra i dati e, soprattutto, di doverle scrivere ad una ad una

c) dà dei tests di buon livello.

A propostito del primo punto, bisogna anche tener presente che lo studio dei dati, in qualche modo, è già stato fatto per poter arrivare alla costruzione del programma e in questo senso ci sembra che TESTGE permetta il recupero di tale studio e sia anche una motivazione ad approfondirlo.

La terza considerazione merita attenzione, in quanto è ovvio che, se la struttura dei dati è descritta in modo approssimativo, i tests generati non potranno essere che di livello mediocre. Ciò a nostro avviso è uno dei maggiori punti di forza di un procedimento di questo tipo per la generazione dei tests in quanto se è vero, com'è vero, che la bontà dei tests è in relazione diretta con la bontà della descrizione dati fornita, il programmatore sarà portato a studiare i dati nel modo più approfondito possibile in quanto ciò gli procurerà dei tests migliori (ed anche, non dimentichiamolo, un programma migliore).

Altro fatto importante da sottolineare, secondo noi, è l'introduzione di un linguaggio per la descrizione dei dati. Infatti, in tal modo, la struttura dei dati diventa in qualche modo elaborabile e quindi controllabile almeno da un punto di vista formale, e, cosa più importante, tale struttura può essere introdotta nel testo del programma e quindi recuperabile ogni qualvolta sia necessario (ad es. in fase di manutenzione).

Ed in fase di manutenzione non solo è recuperabile la struttura dei dati, bensì anche i tests per la prova dell'eventuale modifica intervenuta. In effetti, quello che succede normalmente in fase di modifica o di correzione di un errore è di effettuare dei tests specifici per la modifica intervenuta, senza interessarsi a fondo di quello che succede al resto del programma anche perchè, a quel momento, i tests usati nel primo collaudo del programma sono in genere irreperibili. Inserendo, invece, la struttura dei dati nel testo del programma si possono facilmente riavere con un semplice run di TESTGE tutti i tests utilizzati e quindi verificare che la funzionalità globale del programma sia rimasta intatta dopo la modifica intervenuta.

Qualora si decida di introdurre la struttura dei dati nel testo del programma a scopi documentativi, nulla vieta, se lo si ritiene utile, di descrivere con tale linguaggio anche la struttura dei dati di output e quindi di avere sempre, anche di essa, la descrizione a disposizione per interventi successivi.

Resta infine da sottolineare un ultimo punto, a nostro avviso anch'esso molto importante, riguardo al controllo di qualità del programma da parte dell'estensore delle specifiche, che in genere è persona diversa dal programmatore e responsabile di un'intera applicazione. Con i metodi di testing tradizionali, l'analista non può seguire tutta la casistica dei tests esaminati, in quanto in genere suddivisi in più giochi prova ciascuno dei quali esamina un sottoinsieme dei possibili dati in ingresso.

Con questa tecnica, invece, l'analista è facilitato nel suo controllo di qualità in quanto gli basta esaminare la bontà e completezza della struttura dati utilizzata per la generazione dei tests per rendersi conto della bontà dei tests stessi.

È ovvio che TESTGE può essere utilizzato per programmi realizzati con qualsiasi tecnica, anche con tecniche tradizionali, purchè si dia una descrizione della struttura logica dei dati in input. Anzi, con tecniche tipo Phos e Jackson e con tecniche tradizionali, che non prevedono l'utilizzo dell'unico flusso logico di input, c'è la possibilità di un doppio controllo, in quanto i tests e il programma sono costruiti partendo da punti di avvio diversi.

Uno sviluppo che sin d'ora si può ipotizzare riguarda soprattutto un aiuto al debugging nel senso di dare delle indicazioni sull'output che ci si deve attendere da una serie di tests generati. È chiaro, infatti, che il programmatore, nel costruire la struttura logica del flusso in input, conosce certamente qual è l'output atteso per ogni situazione in input e quindi, con opportune indicazioni inserite nella struttura, può avere indicazioni circa l'output previsto.

## PARTE II — TESTGE: UN ESEMPIO DI USO

### 1. INTRODUZIONE

L'obiettivo di questa seconda parte dell'articolo è quello di mostrare un esempio di utilizzo di TESTGE nella generazione di dati di test per un programma volutamente semplice, al solo scopo di concentrare tutta la nostra attenzione sulle caratteristiche dello strumento.

Partendo dalle specifiche del problema l'esempio presenta:

. la soluzione adottata per il programma, evidenziata dal listing

. i dati necessari a TESTGE per la generazione dei dati campione per il programma in questione

. gli output forniti da TESTGE suddivisi tra
  - outputs di supporto e
  - campioni veri e propri

. i risultati delle prove effettuate coi suddetti campioni.

Passo passo con l'esposizione dei punti di cui sopra si evidenzieranno le caratteristiche pratiche di TESTGE nei vari passaggi.

Come già detto, il problema è volutamente semplice e ciò porta ovviamente ad una certa sproporzione tra il lavoro necessario alla predisposizione dei dati per la creazione di records campione (dati di input e TESTGE) e la complessità dei tests ottenuti.

È chiaro, comunque, che tale divario diminuisce col crescere della complessità del problema.

Si deve notare inoltre che tra i dati in input al problema sono stati inseriti dei campi ad hoc, non utilizzati nel problema stesso, al solo scopo di mostrare una applicazione sufficientemente completa di TESTGE.

Non entreremo in questa nota nel merito della realizzazione del programma, la cui progettazione può essere affrontata con qualsiasi tecnica di programmazione strutturata o anche con tecniche tradizionali. Ciò che ci preme mettere in evidenza è l'utilizzo di TESTGE in un caso particolare. In fig. 1 è riportato lo schema generale della procedura da seguire partendo dalle specifiche per arrivare ad un programma funzionante.

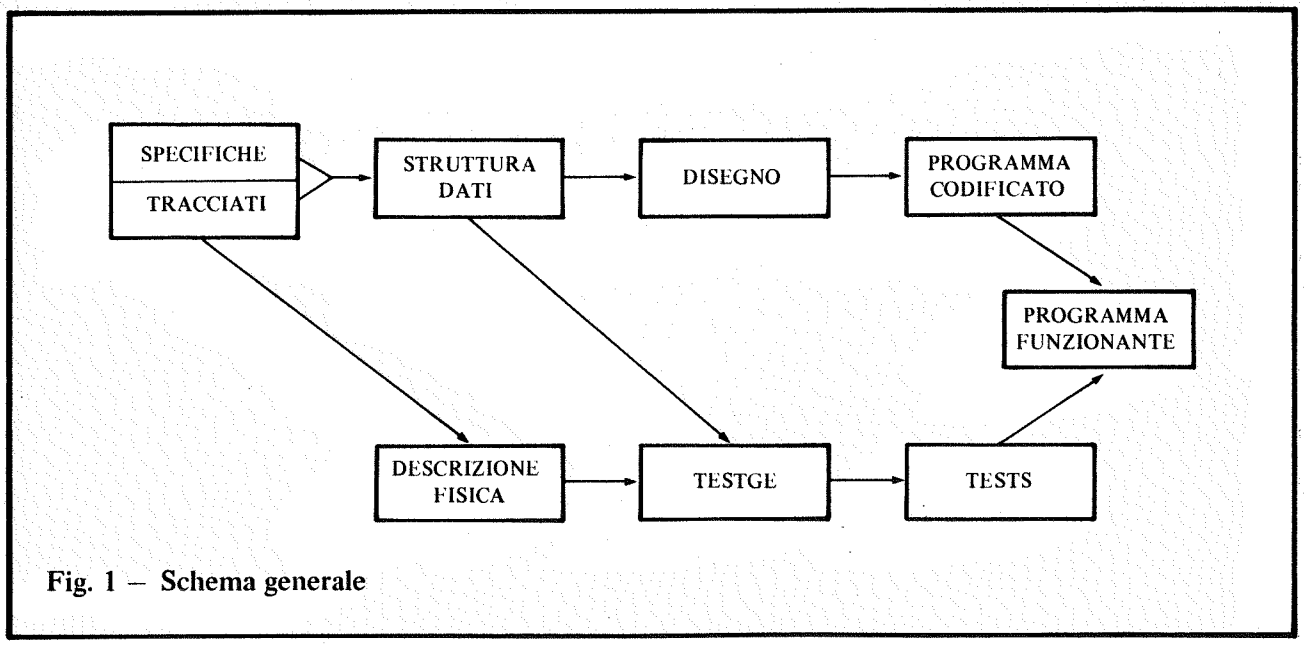



**Fig. 1 — Schema generale**

## 2. SPECIFICHE DEL PROBLEMA

Il problema può essere così sintetizzato (cfr.fig. 2):

INPUT : Flusso ARK, opzionale, ad organizzazione sequenziale ordinato su codice di filiale/codice cliente, comprendente un record inizio filiale (cod. 010), un record per ogni cliente (cod. 020) col relativo saldo.

Flusso MOV ad organizzazione sequenziale ordinato su codice filiale/cliente, riportante i movimenti dare e/o avere di ciascun cliente.

OUTPUT : Flusso ARK aggiornato.

Lista dei soli clienti nuovi e dei clienti con almeno un movimento dare.

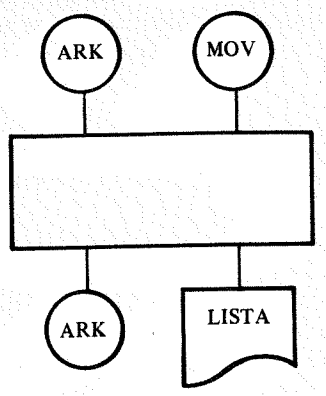



**Fig. 2 – Organigramma e tracciati.**

FLUSSO ARK : INIZIO FIL.

| COD. REC. 010 | COD. FIL. | NOME FILIALE |
|---|---|---|
| X(3) | X(3) | X(12) |

CLIENTE

| COD. REC. 020 | COD. FIL. | COD. CLIENTE | SALDO |
|---|---|---|---|
| X(3) | X(3) | X(6) | 9(12) |

FINE FIL.

| COD. REC. 019 | COD. FIL. | NOME FILIALE | TOTALE FILIALE |
|---|---|---|---|
| X(3) | X(3) | X(12) | 9(12) |

FLUSSO MOV : MOVIM.

| COD. REC. 050 | COD. FIL. | COD. CLIENTE | SEGNO + / – | 0 | MOVIM. |
|---|---|---|---|---|---|
| X(3) | X(3) | X(6) | X | X | 9(10) |

LISTA :

FILIALE DI X ————————X (XXX) – LISTA CLIENTI ANOMALI

| COD. CL | SALDO |
|---|---|
| XXXXXX | 9————9 |
| XXXXXX | 9————9 |

ELABORAZIONE : Aggiornare il flusso ARK inserendo i nuovi clienti, aggiornando il saldo dei clienti movimentati e ricopiando senza modifiche i clienti non movimentati.
Qualora i nuovi clienti appartengano a filiale non presente su ARK, prevedere la creazione degli articoli inizio e fine filiale con nome filiale a blank.

## 3. SOLUZIONE

Diamo in fig. 3 la stesura del programma per ciò che riguarda la PROCEDURE DIVISION senza entrare nel merito della bontà della soluzione.

La tecnica adottata è chiaramente top-down, anche se non segue alla lettera nessuna delle tre metodologie principali (Warnier, Jackson, Phos).

## 4. PREDISPOSIZIONE DATI PER LA GENERAZIONE DEI TESTS

Come si può notare dallo schema riportato nell'introduzione, sono necessari due tipi di dati per la generazione dei tests e precisamente:

- una descrizione fisica di ogni singolo flusso
- una descrizione della struttura logica dell'insieme complessivo di dati di input al programma che si vuole collaudare.

### 4.a DESCRIZIONE FISICA

In figura 4 è riportata con commenti la descrizione fisica dei dati per il nostro programma. Tale descrizione si deduce pressochè interamente dai tracciati dei singoli flussi. (La struttura generale di tale descrizione è riportata nella prima parte dell'articolo).

Come si può vedere, per ogni flusso è prevista almeno una scheda "F" riportante tutti i tipi di record del flusso, per ogni tipo record è prevista almeno una scheda "R" che specifica tipo e lunghezza di ogni singolo campo del record; i singoli campi sono eventualemente specificati da una scheda di tipo "C". Tale scheda è richiesta solo per quei campi di cui è necessario specificare il contenuto o sul cui contenuto sono richieste operazioni particolari (memorizzazione, totalizzazione, ecc.).

In coda alle descrizioni dei flussi e dei relativi records sono richieste alcune descrizioni di chiavi (schede K), di tabelle (schede T), di costanti (schede L), ecc.

Ad esempio, il flusso ARK risulta composto di tre tipi record (010,020,019) e quindi la scheda "F" sarà del tipo:

FARK = 010, 020, 019.

Esaminando ad esempio il tracciato del record 010 possiamo scrivere la relativa scheda "R":

R010 = 010, N03, X12

e quindi per il campo 1:

C001 = CK0, MM1

e cioè il campo 001 (cod. di filiale) deve contenere la chiave 0 (CK0) ed inoltre il suo contenuto deve essere memorizzato nel memorizzatore 1 (MM1).

Qualora il campo di un record possa assumere valori diversi e quindi significati diversi rilevanti per il programma che si sta costruendo, è necessario definire a livello scheda F due o più tipi di records diversi per lo stesso record, ciascuno col suo valore caratteristico. Si veda ad esempio, nel nostro caso, il flusso MOV, per cui so o stati definiti due tipi records diversi 050 e 051 a seconda che si tratti di movimento avere o dare. È ovvio che il record generato sarà sempre uno 050, ma con contenuti diversi sul campo segno. Ciò permette, qualora il programma lo preveda, di creare anche dei records anomali.

Vediamo ora, sempre facendo riferimento al nostro programma, qualche esempio di descrizione chiave, tabella e costante.
Ad es. per la chiave 0 (codice di filiale) abbiamo la scheda

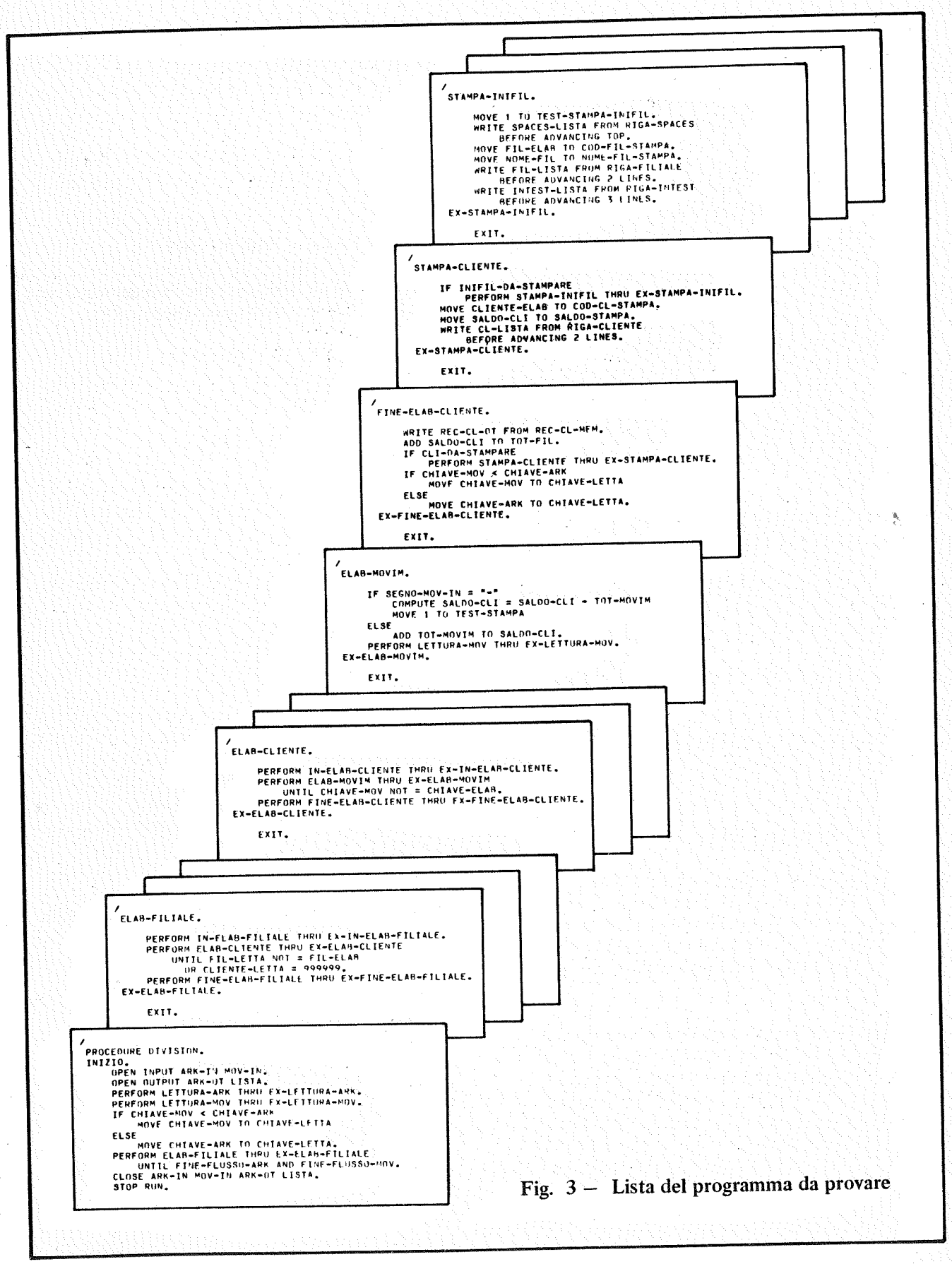
K000 = N03, ST0

che ci dice che la chiave è numerica di 3 caratteri (N03) ed è da ricavare sequenzialmente (S) dalla tabella 0 (T0).

La tabella 0 è specificata dalla scheda

T000 = N03, 010, 010, 020, .....

la quale specifica che è composta di elementi nu-



```
/
 STAMPA-INIFIL.

     MOVE 1 TO TEST-STAMPA-INIFIL.
     WRITE SPACES-LISTA FROM RIGA-SPACES
         BEFORE ADVANCING TOP.
     MOVE FIL-ELAB TO COD-FIL-STAMPA.
     MOVE NOME-FIL TO NOME-FIL-STAMPA.
     WRITE FIL-LISTA FROM RIGA-FILIALE
         BEFORE ADVANCING 2 LINES.
     WRITE INTEST-LISTA FROM RIGA-INTEST
         BEFORE ADVANCING 3 LINES.
 EX-STAMPA-INIFIL.

     EXIT.
```

```
/
 STAMPA-CLIENTE.

     IF INIFIL-DA-STAMPARE
         PERFORM STAMPA-INIFIL THRU EX-STAMPA-INIFIL.
     MOVE CLIENTE-ELAB TO COD-CL-STAMPA.
     MOVE SALDO-CLI TO SALDO-STAMPA.
     WRITE CL-LISTA FROM RIGA-CLIENTE
         BEFORE ADVANCING 2 LINES.
 EX-STAMPA-CLIENTE.

     EXIT.
```

```
/
 FINE-ELAB-CLIENTE.

     WRITE REC-CL-OT FROM REC-CL-WFM.
     ADD SALDO-CLI TO TOT-FIL.
     IF CLI-DA-STAMPARE
         PERFORM STAMPA-CLIENTE THRU EX-STAMPA-CLIENTE.
     IF CHIAVE-MOV < CHIAVE-ARK
         MOVE CHIAVE-MOV TO CHIAVE-LETTA
     ELSE
         MOVE CHIAVE-ARK TO CHIAVE-LETTA.
 EX-FINE-ELAB-CLIENTE.

     EXIT.
```

```
/
 ELAB-MOVIM.

     IF SEGNO-MOV-IN = "-"
         COMPUTE SALDO-CLI = SALDO-CLI - TOT-MOVIM
         MOVE 1 TO TEST-STAMPA
     ELSE
         ADD TOT-MOVIM TO SALDO-CLI.
     PERFORM LETTURA-MOV THRU EX-LETTURA-MOV.
 EX-ELAB-MOVIM.

     EXIT.
```

```
/
 ELAB-CLIENTE.

     PERFORM IN-ELAB-CLIENTE THRU EX-IN-ELAB-CLIENTE.
     PERFORM ELAB-MOVIM THRU EX-ELAB-MOVIM
         UNTIL CHIAVE-MOV NOT = CHIAVE-ELAB.
     PERFORM FINE-ELAB-CLIENTE THRU EX-FINE-ELAB-CLIENTE.
 EX-ELAB-CLIENTE.

     EXIT.
```

```
/
 ELAB-FILIALE.

     PERFORM IN-ELAB-FILIALE THRU EX-IN-ELAB-FILIALE.
     PERFORM ELAB-CLIENTE THRU EX-ELAB-CLIENTE
         UNTIL FIL-LETTA NOT = FIL-ELAB
             OR CLIENTE-LETTA = 999999.
     PERFORM FINE-ELAB-FILIALE THRU EX-FINE-ELAB-FILIALE.
 EX-ELAB-FILIALE.

     EXIT.
```

```
/
 PROCEDURE DIVISION.
 INIZIO.
     OPEN INPUT ARK-IN MOV-IN.
     OPEN OUTPUT ARK-OT LISTA.
     PERFORM LETTURA-ARK THRU EX-LETTURA-ARK.
     PERFORM LETTURA-MOV THRU EX-LETTURA-MOV.
     IF CHIAVE-MOV < CHIAVE-ARK
         MOVE CHIAVE-MOV TO CHIAVE-LETTA
     ELSE
         MOVE CHIAVE-ARK TO CHIAVE-LETTA.
     PERFORM ELAB-FILIALE THRU EX-ELAB-FILIALE
         UNTIL FINE-FLUSSO-ARK AND FINE-FLUSSO-MOV.
     CLOSE ARK-IN MOV-IN ARK-OT LISTA.
     STOP RUN.
```

**Fig. 3 — Lista del programma da provare**

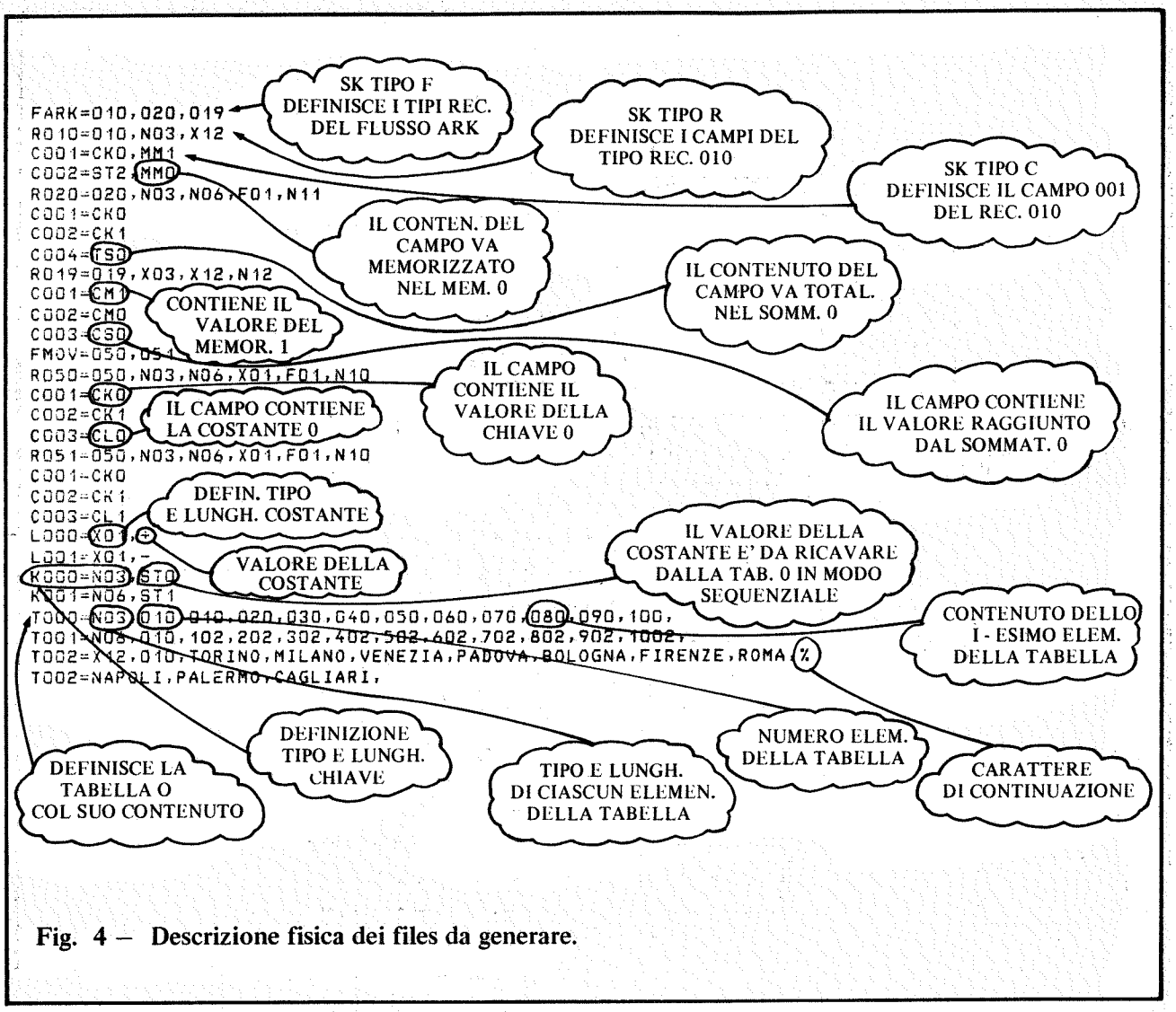



**Fig. 4 – Descrizione fisica dei files da generare.**

merici di 3 caratteri in numero di 10 (primo 010) con di seguito gli elementi della tabella.

La scheda

L000 = X01, +

indica che la costante 0 è di tipo alfanumerico di lunghezza 1 carattere, e costituita dal carattere "+".

## 4.b STRUTTURA LOGICA

La struttura logica è la parte più complessa dei dati di input a TESTGE e per la sua stesura è necessario avere presente sia la struttura di ogni singolo flusso di input che le specifiche del problema.

Per il nostro problema e utilizzando la grafia di Jackson la struttura dei dati è riportata in fig. 5.

Come si vede, i vari accoppiamenti sono esplicitati ai livelli cui si verificano in modo da evitare, nella generazione dei campioni, il proliferare di casi che in realtà sono impossibili.

Ciò a prima vista sembra portare ad una eccessiva ramificazione dell'albero; in realtà, utilizzando una opportuna suddivisione in sottostrutture, ciò può essere evitato, e il lavoro stesso di costruzione semplificato, dovendo limitare la nostra attenzione a sottostrutture più semplici. Nel nostro caso, ad esempio, la struttura CL-MOV è richiamata 3 volte, una delle quali dalla struttura FIL-SOLO-MOV, richiamata a sua volta in due punti, e quindi praticamente ripetuta 4 volte nella struttura complessiva.

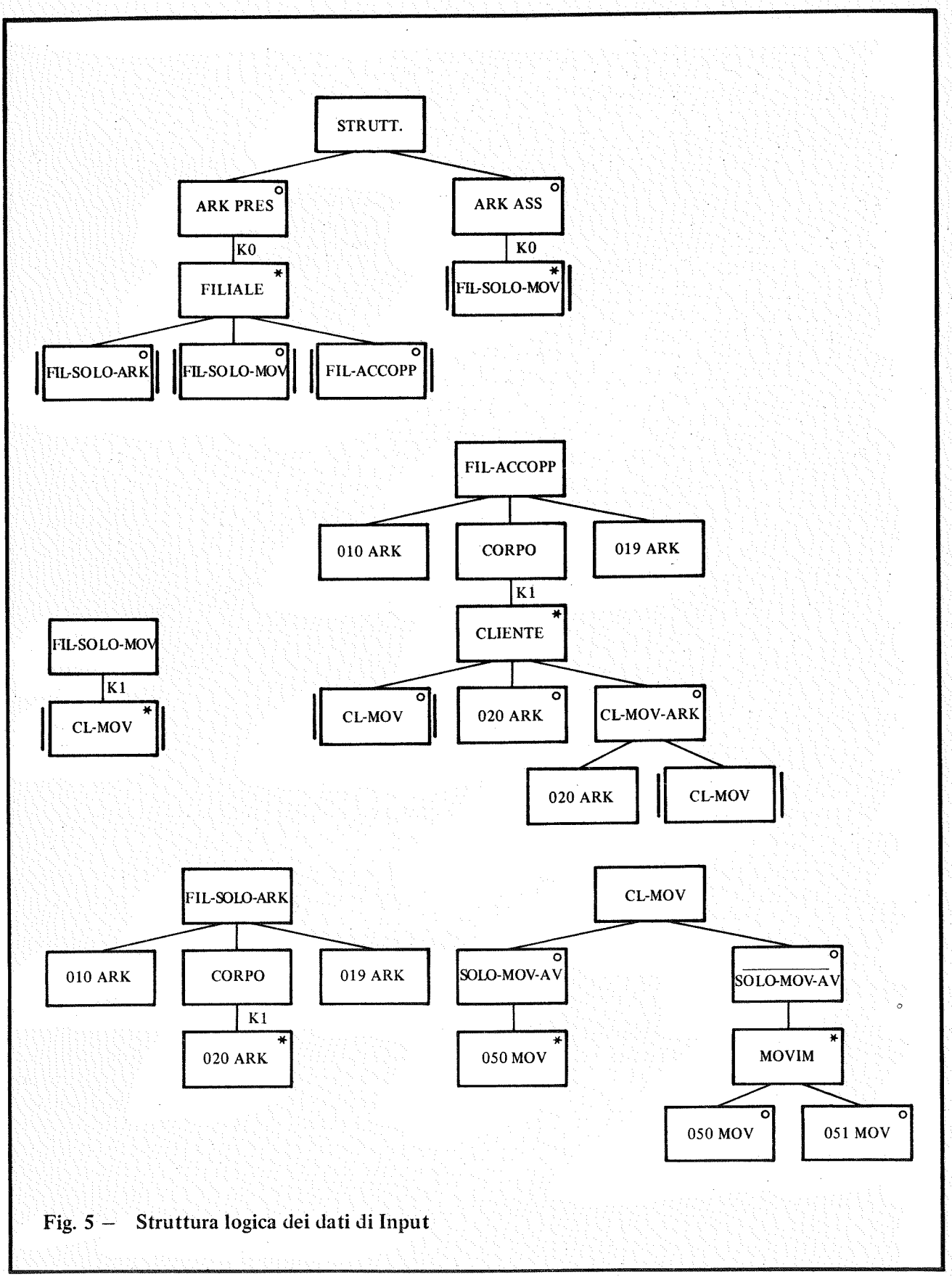
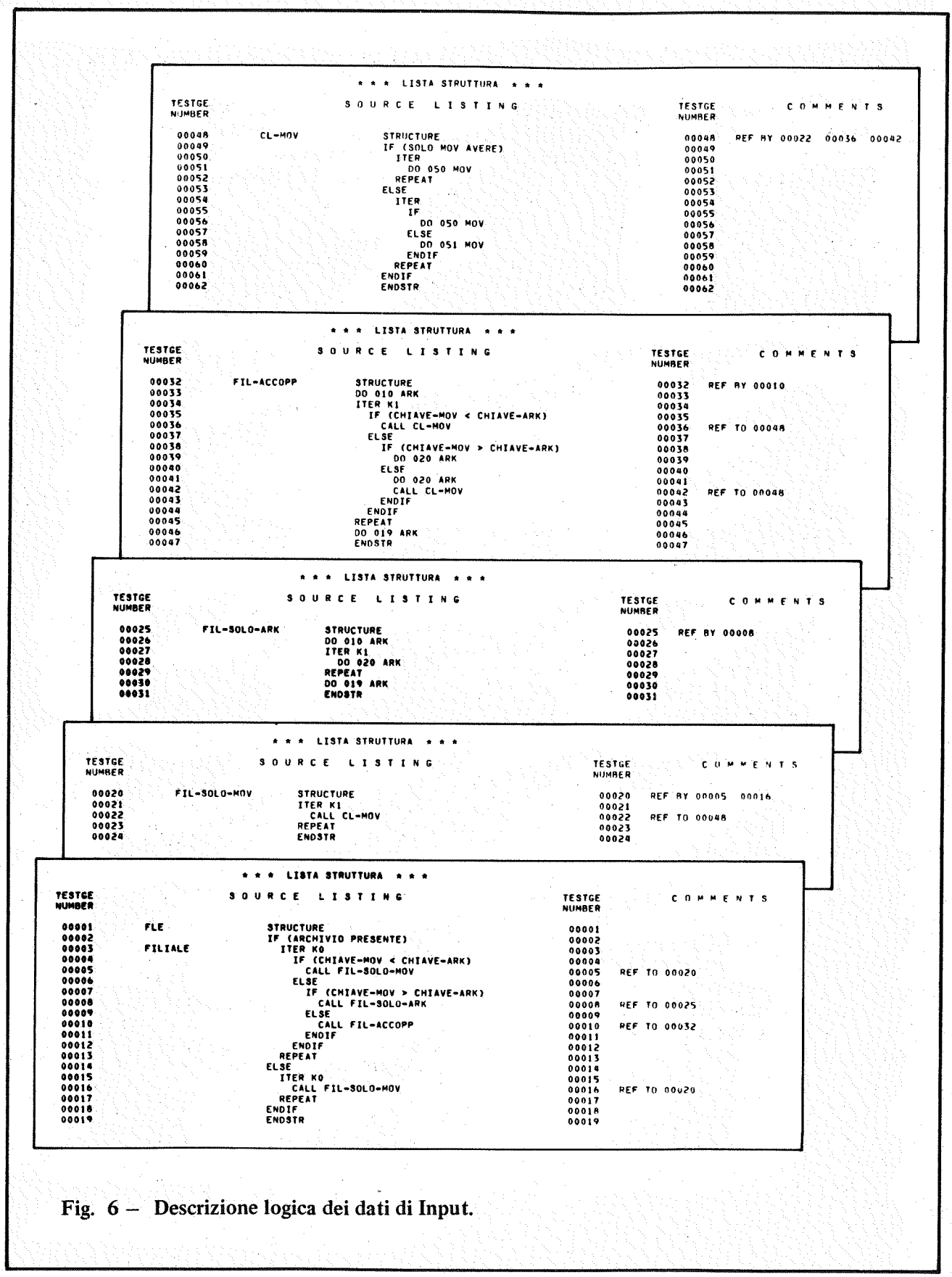



Fig. 5 – Struttura logica dei dati di Input



```
                    * * *  LISTA STRUTTURA  * * *

TESTGE                  S O U R C E   L I S T I N G                    TESTGE          C O M M E N T S
NUMBER                                                                 NUMBER

 00048      CL-MOV          STRUCTURE                                   00048   REF BY 00022  00036  00042
 00049                      IF (SOLO MOV AVERE)                         00049
 00050                        ITER                                      00050
 00051                          DO 050 MOV                              00051
 00052                        REPEAT                                    00052
 00053                      ELSE                                        00053
 00054                        ITER                                      00054
 00055                          IF                                      00055
 00056                            DO 050 MOV                            00056
 00057                          ELSE                                    00057
 00058                            DO 051 MOV                            00058
 00059                          ENDIF                                   00059
 00060                        REPEAT                                    00060
 00061                      ENDIF                                       00061
 00062                      ENDSTR                                      00062
```

```
                    * * *  LISTA STRUTTURA  * * *

TESTGE                  S O U R C E   L I S T I N G                    TESTGE          C O M M E N T S
NUMBER                                                                 NUMBER

 00032      FIL-ACCOPP      STRUCTURE                                   00032   REF BY 00010
 00033                      DO 010 ARK                                  00033
 00034                      ITER K1                                     00034
 00035                        IF (CHIAVE-MOV < CHIAVE-ARK)              00035
 00036                          CALL CL-MOV                             00036   REF TO 00048
 00037                        ELSE                                      00037
 00038                          IF (CHIAVE-MOV > CHIAVE-ARK)            00038
 00039                            DO 020 ARK                            00039
 00040                          ELSE                                    00040
 00041                            DO 020 ARK                            00041
 00042                            CALL CL-MOV                           00042   REF TO 00048
 00043                          ENDIF                                   00043
 00044                        ENDIF                                     00044
 00045                      REPEAT                                      00045
 00046                      DO 019 ARK                                  00046
 00047                      ENDSTR                                      00047
```

```
                    * * *  LISTA STRUTTURA  * * *

TESTGE                  S O U R C E   L I S T I N G                    TESTGE          C O M M E N T S
NUMBER                                                                 NUMBER

 00025      FIL-SOLO-ARK    STRUCTURE                                   00025   REF BY 00008
 00026                      DO 010 ARK                                  00026
 00027                      ITER K1                                     00027
 00028                        DO 020 ARK                                00028
 00029                      REPEAT                                      00029
 00030                      DO 019 ARK                                  00030
 00031                      ENDSTR                                      00031
```

```
                    * * *  LISTA STRUTTURA  * * *

TESTGE                  S O U R C E   L I S T I N G                    TESTGE          C O M M E N T S
NUMBER                                                                 NUMBER

 00020      FIL-SOLO-MOV    STRUCTURE                                   00020   REF BY 00005  00016
 00021                      ITER K1                                     00021
 00022                        CALL CL-MOV                               00022   REF TO 00048
 00023                      REPEAT                                      00023
 00024                      ENDSTR                                      00024
```

```
                    * * *  LISTA STRUTTURA  * * *

TESTGE                  S O U R C E   L I S T I N G                    TESTGE          C O M M E N T S
NUMBER                                                                 NUMBER

 00001      FLE             STRUCTURE                                   00001
 00002                      IF (ARCHIVIO PRESENTE)                      00002
 00003      FILIALE           ITER K0                                   00003
 00004                          IF (CHIAVE-MOV < CHIAVE-ARK)            00004
 00005                            CALL FIL-SOLO-MOV                     00005   REF TO 00020
 00006                          ELSE                                    00006
 00007                            IF (CHIAVE-MOV > CHIAVE-ARK)          00007
 00008                              CALL FIL-SOLO-ARK                   00008   REF TO 00025
 00009                            ELSE                                  00009
 00010                              CALL FIL-ACCOPP                     00010   REF TO 00032
 00011                            ENDIF                                 00011
 00012                          ENDIF                                   00012
 00013                        REPEAT                                    00013
 00014                      ELSE                                        00014
 00015                        ITER K0                                   00015
 00016                          CALL FIL-SOLO-MOV                       00016   REF TO 00020
 00017                        REPEAT                                    00017
 00018                      ENDIF                                       00018
 00019                      ENDSTR                                      00019
```

**Fig. 6 – Descrizione logica dei dati di Input.**

Dalla struttura così descritta con grafia Jackson si passa abbastanza semplicemente alla struttura descritta per TESTGE come riportato in fig. 6.

In realtà tale figura riporta la struttura dei dati così come viene editata da TESTGE, comunque essa differisce dall'input unicamente per l'inserimento del numero progressivo e dei riferimenti.

## 5. OUTPUT DI TESTGE

Possiamo distinguere gli outputs di TESTGE tra

- outputs di supporto e
- tests.

### 5.a OUTPUTS DI SUPPORTO

Come outputs di supporto abbiamo innanzitutto la struttura logica dei dati in input di cui si è già parlato nel paragrafo precedente (fig. 6). Tale struttura è corredata di rimandi, e precisamente le schede STRUCTURE danno l'indirizzo del punto da cui sono richiamate e per ogni scheda CALL viene indicata la posizione della struttura richiamata. In futuro si pensa di introdurre altri riferimenti per i records da generare, in modo da poter individuare facilmente nella struttura, per ogni record, il punto (o i punti) in cui viene generato.

Oltre alla struttura, viene prodotta una mappa dei giochi prova generati (vedi fig. 7) che riporta il numero dei giochi prova e, per ogni gioco prova, il numero di records effettivi generati su ciascun flusso con l'indicazione della posizione del record fittizio iniziale e finale. Ciò permette all'utilizzatore di estrarre dai flussi generati il gioco prova effettivo per il testing.

Nel nostro caso era abbastanza semplice prevedere che i giochi prova sarebbero stati almeno due, essendo il flusso ARK opzionale. Infatti si vede che nel secondo gioco prova per ARK non è stato generato alcun record effettivo.

### 5.b TESTS

I tests generati sono riportati in fig. 8 e fig. 9.

Come si vede, per i due flussi (ARK e MOV) ci sono due giochi prova generati: ognuno inizia con un record di fine gioco prova (tipo record ***) e termina con un record di fine gioco prova (tipo record ###).

Per il flusso ARK in particolare il secondo gioco prova non contiene alcun record effettivo e i due records inizio e fine sono successivi.

## 6. RISULTATI DELLE PROVE

A questo punto non ci resta che esporre i risultati delle due prove richieste dai nostri campioni generati da TESTGE e verificare che i risultati sono quelli attesi.

L'estrazione dei tests dai flussi generati è stata fatta mediante UTILITY utilizzando le informazioni date dalla mappa.

In figura 10 sono riportati i risultati commentati del primo gioco prova.

## BIBLIOGRAFIA


- M. A Jackson, Principles of Program Design, Academic Press, 1975
- J. D. Warnier, Introduzione alla programmazione, vol. I (La costruzione dei programmi e vol. II (L'elaborazione dei dati), Etas Libri, 1974
- AA. VV., PHOS: una metodologia per la costruzione veloce ed affidabile dei programmi, NOTE DI SOFTWARE n. 4, Ottobre - Dicembre 1977
- AA. VV., PHOS: una applicazione in ambiente E.D.P., ibidem
- G.J. Myers, Reliable Software Through Composite Design, Petrocelli/Charter, 1975
- R. Rustin, Debugging Techniques in Large Systems, Prentice-Hall

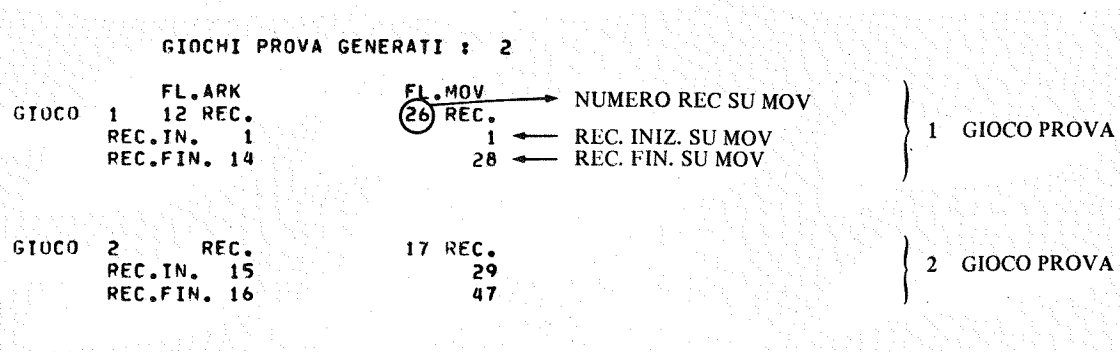



**Fig. 7 – Prospetto riassuntivo giochi prova generati.**

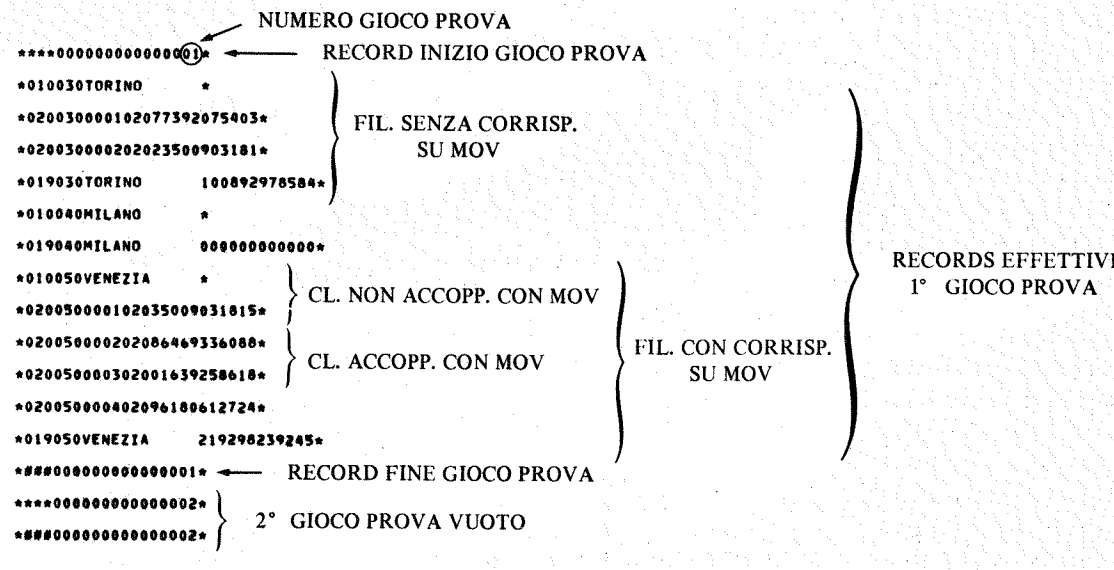



**Fig. 8– Flusso ARK generato.**

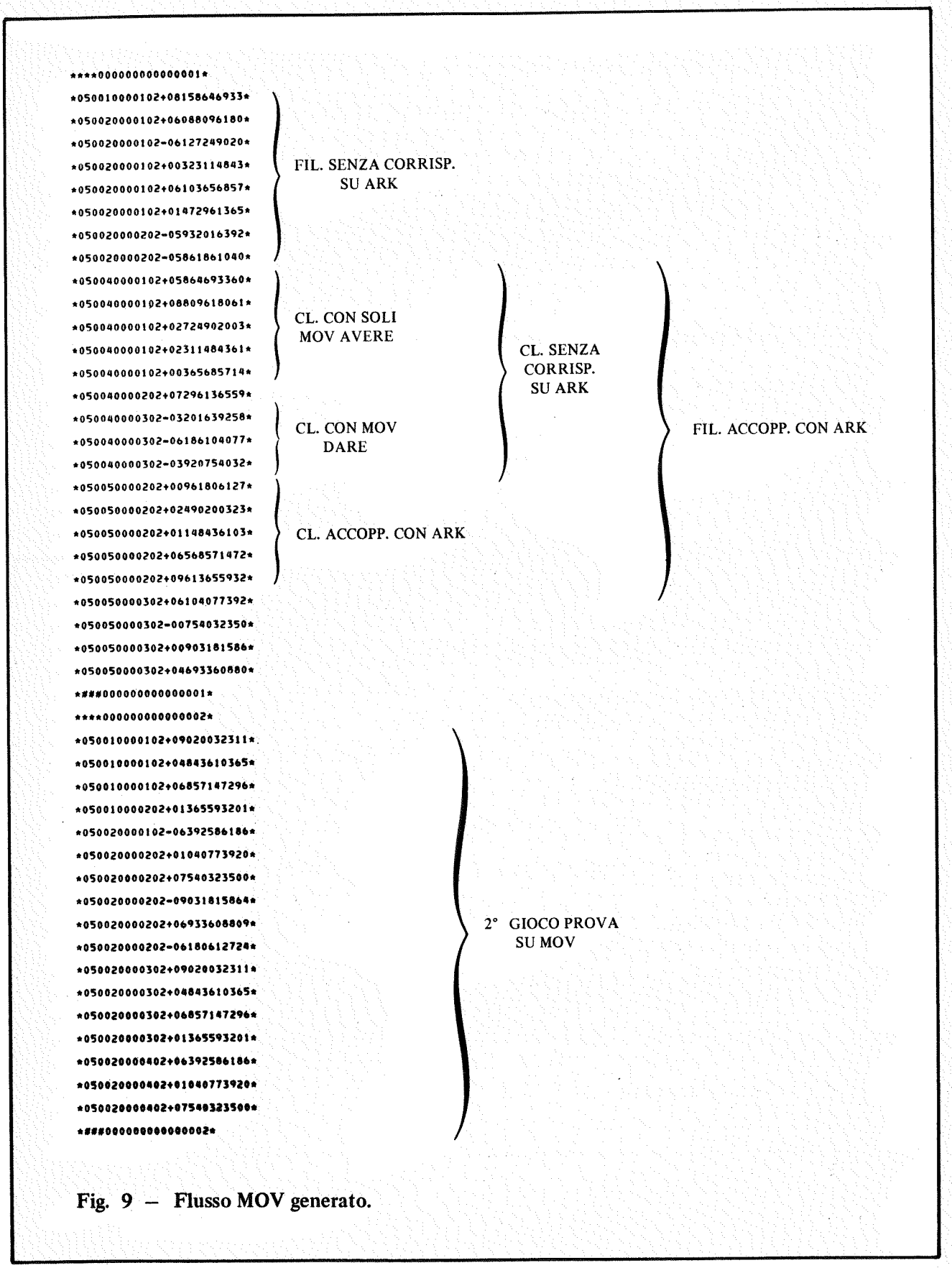



**Fig. 9 – Flusso MOV generato.**

```
*010010          *
*020010000102008158646933*
*019010          008158646933*
*010020          *
*020020000102007860580225*
*020020000202011793877432K*
*019020          00393329720P*
*010030TORINO    *
*020030000102077392075403*
*020030000202023500903181*
*019030TORINO    1008929785B4*
*010040MILANO    *
*020040000102020076383499*
*020040000202007296136559*
*020040000302013308497367P*
*019040MILANO    014064022691*
*010050VENEZIA   *
*020050000102035009031815*
*020050000202107252066045*
*020050000302012585846126*
*020050000402096180612724*
*019050VENEZIA   251027496710*
```

FIL. NUOVE PROVEN. DA MOV

FIL. RICOPIATA DA ARK IN INGRESSO

FIL. AGGIORNATE SULLA BASE DEI MOVIMENTI DI MOV

a) Flusso ARK aggiornato

```
FILIALE DI          (010) - LISTA CLIENTI ANOMALI
     COD.CL.             SALDO
     000102         8158646933

FILIALE DI          (020) - LISTA CLIENTI ANOMALI
     COD.CL.             SALDO
     000102          7860580225
     000202         11793877432

FILIALE DI MILANO   (040) - LISTA CLIENTI ANOMALI
     COD.CL.             SALDO
     000102         20076383499
     000202          7296136559
     000302         13308497367

FILIALE DI VENEZIA  (050) - LISTA CLIENTI ANOMALI
     COD.CL.             SALDO
     000302         12585846126
```

FIL. NUOVA

FIL. NUOVA

CLIENTI NUOVI

CL. VECCHIO CON MOV. DARE

b)Tabulato prodotto

Fig. 10 – Output del programma in prova sulla base del 1° gioco prova

# LE RETI DI PETRI NELLA DESCRIZIONE DI PROCEDURE ORIENTATE ALLA PRODUZIONE DI SOFTWARE

G. Degli Antoni*, D. Marini*, M. Parini°*, B. Zonta °°*

**Riassunto**

Lo scopo di questa nota è mostrare l'impiego delle reti di Petri per la descrizione di procedure di produzione del software. Dopo una breve presentazione delle reti di Petri, l'uso del formalismo per la descrizione di procedure viene illustrato appoggiandosi a un semplice esempio, tratto da un ambiente di produzione di software.

**Abstract**

The purpose of this paper is to show how Petri Nets can be used to describe software production procedures. After a short introduction to the concepts and notations of Petri Nets, the paper develops a simple example, typical of a software production environment.

## 1. INTRODUZIONE

La produzione di software è una attività complessa che poco si presta a essere svolta secondo schemi di procedure rigide; conseguentemente richiede azioni di gestione tendenti a rendere compatibile lo stato del processo con le previsioni e le norme della procedura adottata.

La ragione di ciò va innanzitutto ricercata proprio nella complessità della attività di produzione del software e quindi nella difficoltà di fornire una descrizione della procedura produttiva per la mancanza di adeguate notazioni.

D'altra parte la disponibilità di una notazione per la descrizione di procedure produttive fornisce la base per lo studio delle strutture dei dati atte a rappresentare le stesse e quindi la base per un modo di studiare i problemi connessi con la meccanizzazione delle attività di gestione relative alla produzione del software, o comunque con la gestione di processi produttivi.

* Istituto di Cibernetica dell'Università degli Studi di Milano

° Honeywell Information Systems Italia

°° CNR - Centro di Cibernetica e Attività Linguistiche -Milano

Il presente lavoro è stato svolto nel quadro del progetto CP di collaborazione tra HISI e Istituto di Cibernetica dell'Università degli Studi di Milano.

È appunto scopo di questa nota il mostrare l'impiego di formalismi per descrivere una procedura di produzione del software sperimentata nei laboratori di Pregnana.

Il formalismo adottato è quello delle Reti di Petri, che viene generalmente suggerito per la descrizione di processi concorrenti in cui possono manifestarsi problemi conseguenti alla condivisione di risorse (conflitti, dead-lock, ecc.).

Lo spunto è fornito da un lavoro di Holt ( 1 ), che impiegò le Reti di Petri per la descrizione di procedure giuridiche, e da un lavoro di Hack ( 2 ), che le ha applicate alla descrizioni di processi produttivi.

Dopo una breve presentazione delle Reti di Petri con una nomenclatura adattata alla descrizione di procedure produttive, si descrive direttamente la procedura in esame.

Si discute brevemente la possibilità e la difficoltà di utilizzare la descrizione formale proposta come modello di riferimento per la gestione del processo produttivo durante il suo svolgersi. Si analizzano gli strumenti gestionali necessari e i rapporti fra la descrizione della procedura produttiva e i programmi per la eventuale meccanizzazione della attività produttiva. Si mostra come la conoscenza delle Reti di Petri fornisca un valido strumento per la definizione delle caratteristiche di tali programmi.

Nel caso della procedura introdotta, si discute la possibilità di individuare proprietà delle Reti di Petri che identifichino caratteristiche interessanti del processo produttivo analizzato dal punto di vista degli obiettivi produttivi.

Si discute infine il problema della descrizione in linguaggio naturale della procedura, mostrando come si possa effettuare una traduzione della Rete di Petri in un testo in lingua naturale, comprensibile a chi non conosca le Reti di Petri o comunque non sia disponibile a impiegare formalismi (apparentemente) complessi.

## 2. RETI DI PETRI E PROCESSI PRODUTTIVI

La situazione esemplificativa scelta consiste nella realizzazione di un programma da parte di due

programmatori A e B, che devono disporre, oltre che di ore-macchina, anche delle specifiche del programma da realizzare. A e B dovranno inoltre sviluppare la documentazione per l'utente del programma e la documentazione dei due moduli di cui e composto.

Nella fig. 1 si rappresenta una situazione come quella descritta: la realizzazione del programma avviene a partire dalla disponibilità di tutte le risorse necessarie e produce il programma con la relativa documentazione, rendendo nuovamente disponibili le risorse utilizzate.

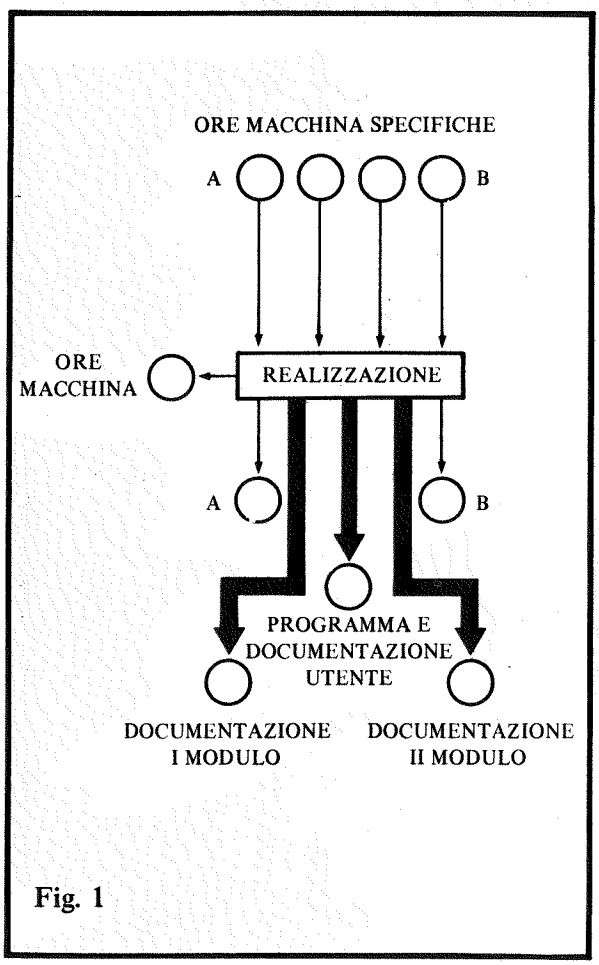



Fig. 1

La schematizzazione di un processo produttivo che è stata adottata e rappresentata in fig. 1 consiste nel considerarlo come un insieme di attività, che, date certe risorse, producono certi prodotti. Nello schema abbiamo:

. **risorse** per la realizzazione del programma:
programmatore A
programmatore B
ore macchina
specifiche del programma

. **attività** di realizzazione del programma con impiego delle risorse

. **prodotti** della realizzazione del programma:
programma documentato
documentazone del I modulo
documentazione del II modulo
rilascio della risorsa A
rilascio della risorsa B
rilascio della risorsa ore-macchina.

Le relazioni temporali e causali che si instaurano tra queste categorie di oggetti possono venire rappresentate con un linguaggio formale, le Reti di Petri, dotato di una rappresentazione grafica come quella vista in fig. 1.

Le Reti di Petri sono grafi orientati bipartiti, dei grafi cioè dotati di due tipi di nodi: ○ e ▭ ( chiamati rispettivamente **posti** e **transizioni** oppure **condizioni** ed **eventi** ), collegati tra loro da archi orientati.

Senza riportare le definizioni formali delle reti di Petri, che possono essere trovate altrove (ad es., 4), qui di seguito vogliamo mettere in luce la capacità di modellare processi propria di questo linguaggio.

Per modellare un processo produttivo assoceremo:

le **attività** agli **eventi**
le **risorse** o i **prodotti** alle **condizioni.**

I legami causali e temporali che correlano risorse e prodotti con le attività che li coinvolgono vengono rappresentati mediante frecce orientate tra le condizioni e gli eventi (le frecce disegnate in grassetto concernono in particolare i prodotti).

In un processo produttivo le risorse o i prodotti possono essere o possono non essere, in un dato momento, disponibili; una determinata attività può dunque essere compiuta se le risorse ad essa necessarie sono in quel momento disponibili, altrimenti occorrerà attendere finchè tale situazione non è verificata. La disponibilità delle risorse o dei prodotti può venire rappresentata nelle Reti di Petri disegnando un grosso punto **(gettone)** nei posti desiderati; si dirà in tal caso che la condizione marcata con il gettone è verificata. Una attività potrà essere eseguita se tutte le sue risorse sono disponibili, nel linguaggio delle Reti di Petri ciò significa che un evento può accadere o "**scattare**" se tutte le sue condizioni di ingresso sono verificate; parleremo in tale situazione di evento abilitato.

Introducendo un concetto di stato nelle Reti di Petri è possibile ottenere una rappresentazione dina-

mica di un processo. Uno stato di una Rete di Petri è l'insieme delle condizioni in un dato istante verificate. Quando una Rete si trova in un determinato stato gli unici eventi che possono scattare sono quelli abilitati, ovvero quelli le cui condizioni di ingresso sono tutte verificate. Questo concetto di stato consente di rappresentare lo stato nel senso usuale di un processo produttivo, che è caratterizzato dai prodotti fino a un dato istante realizzati e dalle risorse in quell'istante disponibili. L'evoluzione del processo consisterà nello svolgimento di tutte quelle attività le cui risorse sono disponibili, realizzando dei prodotti e rilasciando alcune risorse. Nel linguaggio delle Reti questo sviluppo dinamico del processo consiste nella evoluzione della Rete, che, a partire da un determinato stato, provoca lo scattare di tutti gli eventi abilitati e la rimozione dei gettoni dalle condizioni di ingresso degli eventi e la comparsa di un gettone in tutte le condizioni di uscita dagli eventi abilitati. La nuova situazione che si viene a creare è un nuovo stato, che potrà oppure no abilitare nuovi eventi, consentendo o una ulteriore evoluzione della Rete o la fine del processo. Nella fig. 2 sono rappresentati due diagrammi che mostrano gli stati del processo modellato e la sua evoluzione nel linguaggio delle Reti di Petri.

La scelta delle Reti di Petri come strumento per modellare processi produttivi si rivela particolarmente adeguata data la capacità di tale linguaggio di rappresentare processi concorrenti e processi paralleli. Del resto è proprio per modellare processi di tale tipo che le Reti di Petri sono state sviluppate, trovando larga applicazione nllo studio dei sistemi di elaborazione automatica.

Consideriamo concorrenti due processi che si trovano a condividere in maniera antagonistica alcune risorse. Una tipica situazione di questo genere è schematizzata in forma astrata in fig. 3, in cui il verificarsi della condizione p1 abilita gli eventi t1 e t2, dei quali uno solo può scattare, dato che, quando ciò avviene, il gettone di ingresso viene rimosso disabilitando l'altro evento. Quando ciò avviene il processo si troverà in uno dei due stati alternativi a) o b).

Nell'esempio di processo produttivo che stiamo analizzando si può rappresentare con una Rete di Petri una situazione concorrente come la seguente: "il programma e la documentazione non appena disponibili dovranno essere consegnati, oppure aggiornati qualora vengano richieste delle modifiche delle specifiche iniziali". La situazione è schematizzata in fig. 4, in cui abbiamo rappresentato il

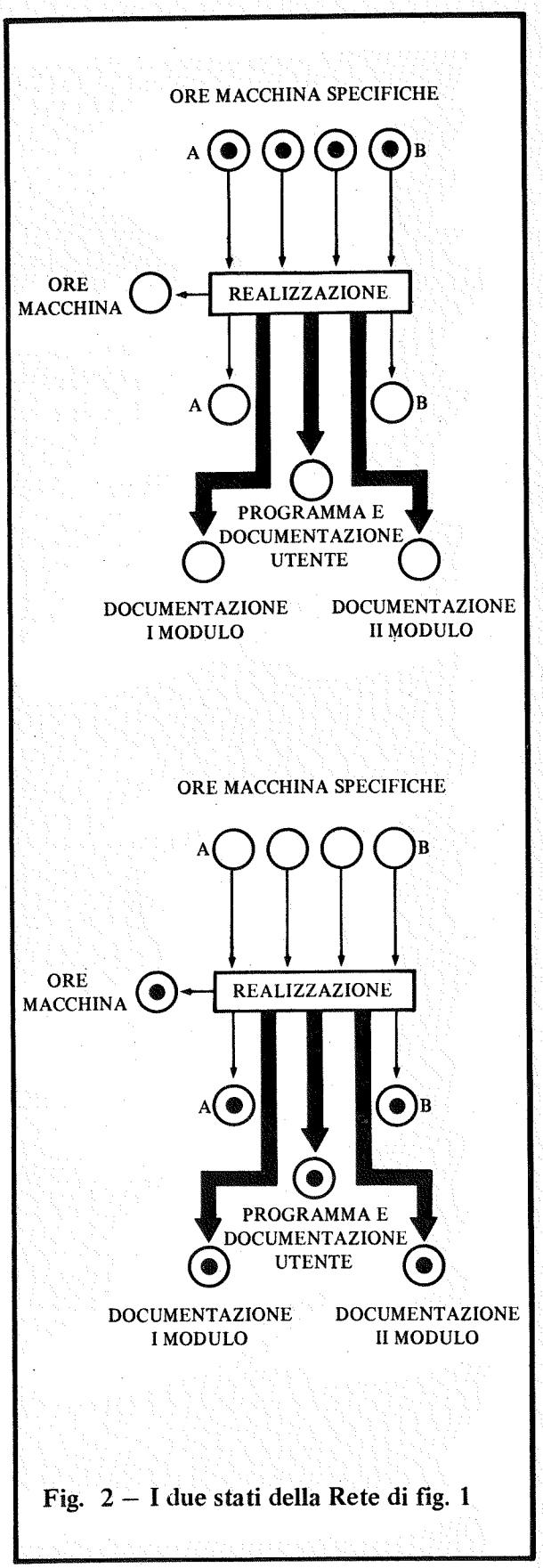



**Fig. 2 – I due stati della Rete di fig. 1**

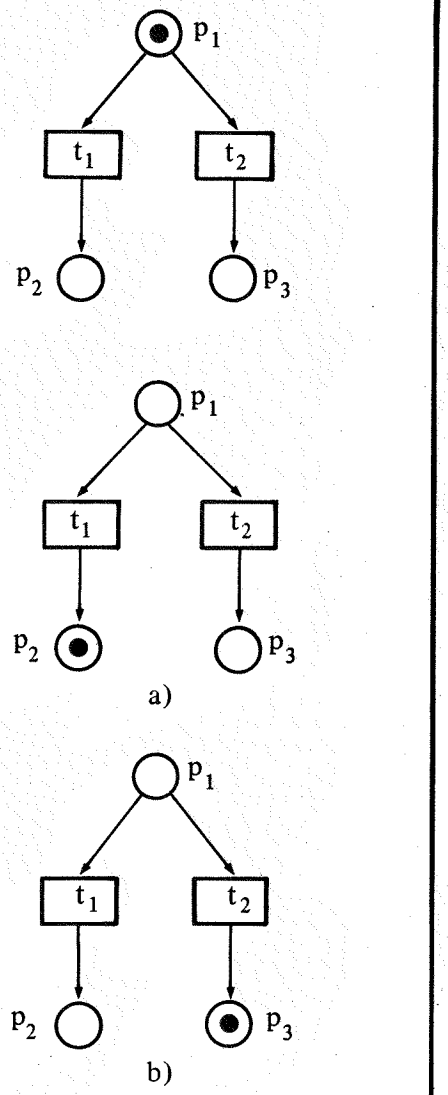



**Fig. 3 – Conflitto o concorrenza**

processo di produzione del programma nella sua globalità, anche se ancora in forma molto generale; la Rete si trova nello stato caratterizzato dalla presenza di una richiesta di modifiche(la freccia tratteggiata in ingresso alla condizione suddetta serve a denotare la provenienza di tale richiesta da un altro processo che non è modellato).

Il carattere conflittuale di tale situazione consiste nel dover operare una scelta su quale delle due attività possibili debba utilizzare le risorse disponibili (documentazione, programmatori, programmi, ore-macchina ecc.). Infatti gli eventi "consegna" ed "aggiornamento" possono entrambi accadere in quanto le loro condizioni di ingreso sono tutte verificate. La Rete non consente di modellare in modo deterministico quale dei due possibili eventi debba necessariamente accadere, e questo, se da una parte limita la capacità espressiva di questo linguaggio, dall'altra ne accresce la semplicità.

Come vedremo meglio più oltre, questo fatto rende particolarmente adatte le Reti di Petri come supporto per i processi decisionali propri della gestione della produzione.

Proseguendo l'analisi della situazione esemplificativa specifichiamo come può essere svolta la complessa attività di "realizzazione"; emergerà da ciò il problema del parallelismo, ovvero dello svolgersi contemporaneo dei due processi indipendenti, che possono tuttavia in alcune situazioni condividere le stesse risorse o utilizzare l'uno i prodotti dell'altro.

Notiamo che una situazione di parallelismo viene rappresentata con Reti di Petri come in fig. 5, in cui l'evoluzione del processo dallo stato a) allo stato b) avvia due processi che si potranno sviluppare in seguito contemporaneamente.

Tornando all'esempio: data l'indipendenza dei due moduli, il loro sviluppo può essere realizzato parallelamente dai due programmatori, che si ripartiscono le ore macchina in modo opportuno. Ciascuna realizzazione comporta: la definizione della struttura del programma, lo sviluppo della documentazione, lo studio del modulo, la codifica, la compilazione e la correzione di errori formali e il testo. Completato lo sviluppo dei due moduli, essi vengono integrati e provati (eventualmente corretti e modificati insieme con la rispettiva documentazione), e viene prodotta la documentazione per l'utente. In fig. 6 è rappresentata una prima espansione dettagliata della attività di realizzazione, che evidenzia il parallelismo.

Nello schema di fig. 7 l'insieme delle attività è rappresentato in dettaglio mettendo in evidenza il modo in cui la risorsa ore-macchina viene condivisa dai due programmatori (per semplificare la rappresentazione non abbiamo evidenziato le risorse: programmatori A e B). Questo aspetto merita una qualche attenzione (nel diagramma sono disegnati in grassetto gli archi e le condizioniche riguardano le ore-macchina); quando A o B intendono compilare il loro programma richiedono ore-macchina, e per tutta la durata della complitazione tale risorsa

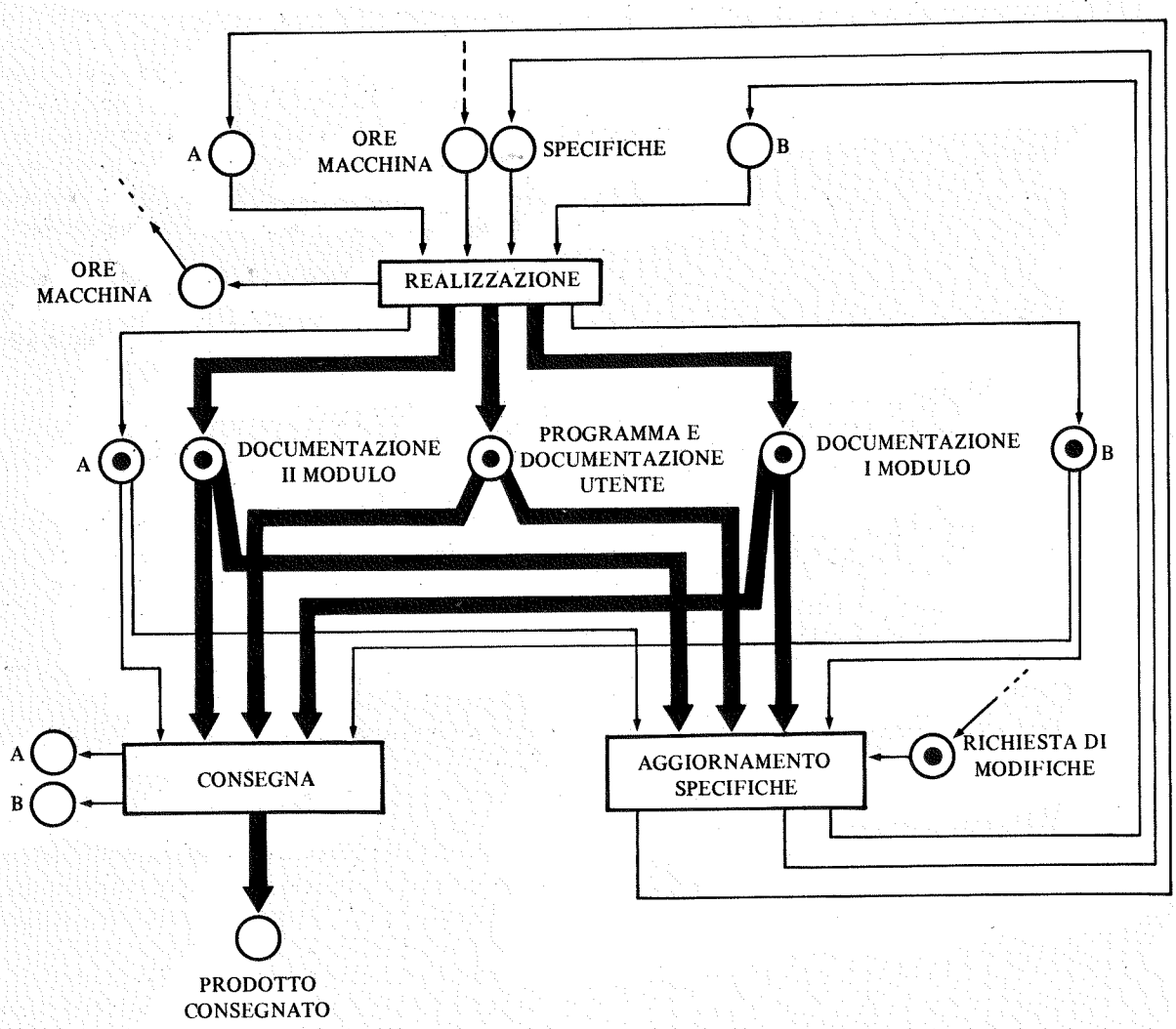



**Fig. 4 – Il processo di sviluppo di un programma con la possibilità di richiesta di modifiche: una situazione concorrente.**

non è disponibile.
Completata la compilazione la risorsa diviene nuovamente disponibile sia per la compilazione dell'altro modulo che per nuove compilazioni al procedere del debugging. La situazione che si realizza è ancora di conflitto o concorrenza tra i due programmatori, che può venire risolta con interventi "esterni", cioè non rappresentati nella rete disegnata.

## 3. LE RETI DI PETRI COME SUPPORTO ALLA GESTIONE

Si è cercato di illustrare come le Reti di Petri costituiscano una notazione adeguata a rappresentare un processo di produzione di software. Vogliamo ora discutere la possibilità di utilizzare questa notazione come modello di riferimento per la gestione dell'attività di produzione del software e come base per lo sviluppo di strumenti automatici di supporto alla gestione.

La gestione di un processo produttivo è un complesso di attività che comportano numerosi momenti decisionali, destinati a realizzare un coordinamento ottimale dei differenti processi individuali per il raggiungimento di un obiettivo desiderato. Ogni decisione, per poter essere adeguatamente compiuta, richiede la conoscenza dello stato in cui il processo si trova e la conoscenza delle conseguenze, prevedibili o previste, delle decisioni possibili.

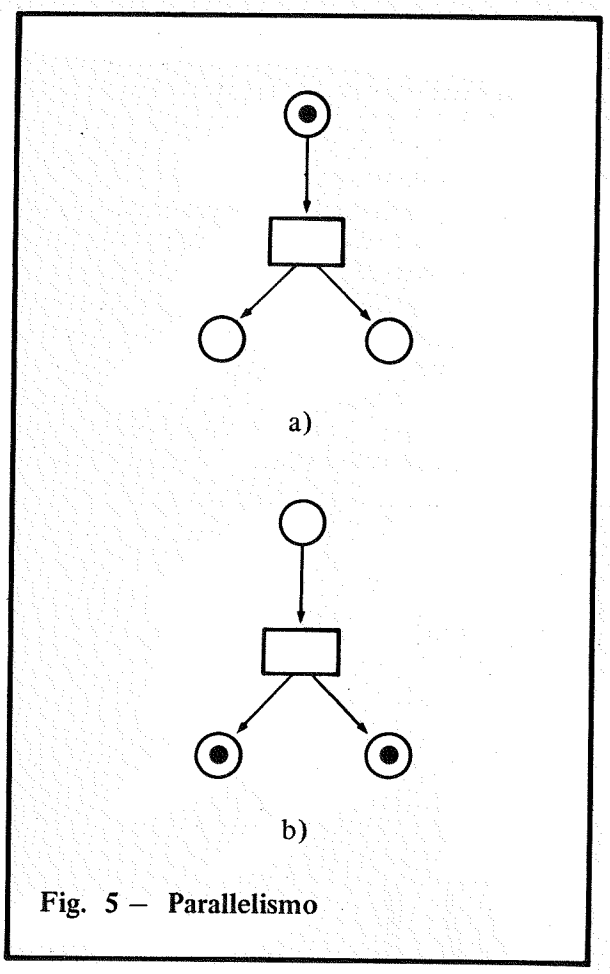



**Fig. 5 – Parallelismo**

Assumendo come rappresentazione di un processo produttivo una Rete di Petri, due sono gli aspetti su cui si esercita l'attività decisionale-gestionale che non sono stati esplicitamente modellati con la Rete dell'esempio: la risoluzione dei conflitti e la durata temporale delle attività. D'altra parte la Rete evidenzia, manifestando una particolare struttura, l'esistenza di conflitti e la loro localizzazione nel complesso del processo; si può pertanto ipotizzare che il gestore-decisore possa utilizzare la Rete di Petri come riferimento costante, come schema al quale ritornare ogni volta che egli ritenga di dovere compiere una verifica dello stato del processo o sia consapevole della necessità di un proprio intervento risolutore. La lettura in ogni momento della Rete evidenzia dove e quando si pongono situazioni di conflitto, da quali fattori esse dipendono (nel linguaggio delle Reti: quali risorse o prodotti intermedi abilitano gli eventi in conflitto), a quali processi daranno luogo (nel linguaggio delle Reti: quali saranno i prodotti successivi agli eventi in conflitto). Per illustrare come un problema di questo genere non presenti sempre soluzioni semplici, ma spesso dipenda da altri fattori esterni al processo, la cui modellazione o renderebbe incomprensibile la Rete o sarebbe impossibile, data la quantità di informazioni da cui può dipendere, consideriamo un semplice esempio. Si debba ripartire la risorsa ore-macchina tra due programmatori che devono compilare due moduli sorgente, e uno dei due debba successivamente utilizzare il modulo sviluppato dall'altro programmatore per integrarlo con un terzo modulo, che deve essere stato nel frattempo documentato (cifr. fig. 8).

La rappresentazione in fig. 8 della situazione conflittuale già evidenzia con più dettaglio dove debba esercitarsi l'azione gestionale-decisionale. Perchè questa possa esplicarsi, tuttavia, sono necessarie altre informazioni, in particolare quelle concernenti la durata temporale delle singole attività.
Questa può venire esplicitata ipotizzando che lo sviluppo del primo modulo richieda 10 ore, quello del secondo 5 e 5 ore la documentazione del terzo. Le due possibili soluzioni al conflitto provocheranno o la fermata di B per 5 ore, mantenendo A impegnato, oppure la fermata di A per 5 ore mantenendo B impegnato.

La scelta di quale delle due soluzioni sia preferibile non è più riconducibile a fattori rappresentati nella Rete, dipende piuttosto da fattori esterni, che riguardano altri processi produttivi a cui i soggetti coinvolti in quello in esame possono partecipare. È tuttavia indubbio che in tale modo si è posta in evidenza la presenza del conflitto e le conseguenze delle possibili risoluzioni.

Possiamo ora svolgere alcune considerazioni di carattere generale, tese a valutare la validità del linguaggio delle Reti di Petri per supportare l'attività gestionale.

Associando una durata temporale alle attività la Rete di Petri può essere analizzata come una rete PERT ( 3 ), per identificarne i cammini critici; a differenza delle reti PERT, tuttavia, le Reti di Petri evidenziano due importanti aspetti: primo, dove sono localizzati i momenti decisionali nel processo produttivo; secondo, come vengono condivise, nel corso del processo, le risorse.

Lo schema presentato come esempio nella fig. 8 può essere considerato come il risultato di una indagine che è stata svolta a monte dell'avvio del processo produttivo e che ha avuto come esito una particolare attribuzione delle attività ai due programmatori. Si può ovviamente ipotizzare una differente attribuzione delle attività, a ciascuna delle quali corrispondono differenti processi produttivi

e loro differenti rappresentazioni. Ogni rappresentazione può allora diventare oggetto di studio: tramite simulazione, manuale o automatizzata, dell'evoluzione del processo si può cercare di prevedere quale sia, secondo criteri opportuni, la struttura del processo ottimale prima che esso si avvii.

Riassumendo, il supporto che le Reti di Petri possono fornire nella attività gestionale dei processi di produzione del software può esercitarsi in almeno due fasi: nella fase di definizione e scelta della struttura ottimale del processo produttivo, e nella fase di gestione del processo stesso, ogni volta che sia necessario compiere azioni di verifica dello stato del processo e dell'andamento della produzione o sia necessario risolvere situazioni di conflitto o prendere decisioni sulla durata stessa di singole attività.

Nella prima fase si possono esplorare le differenti ipotesi e, disponendo di supporti automatici interattivi, simulare l'evoluzione dei differenti processi ipotizzati, associando alle attività una durata ed evidenziando i cammini critici, identificando i momenti decisionali e valutando i risultati di differenti possibili scelte. Nella seconda fase, scelta una struttura per il processo, lo si può mantenere costantemente sotto controllo, verificando scostamenti temporali dalle scadenze previste, segnalan-

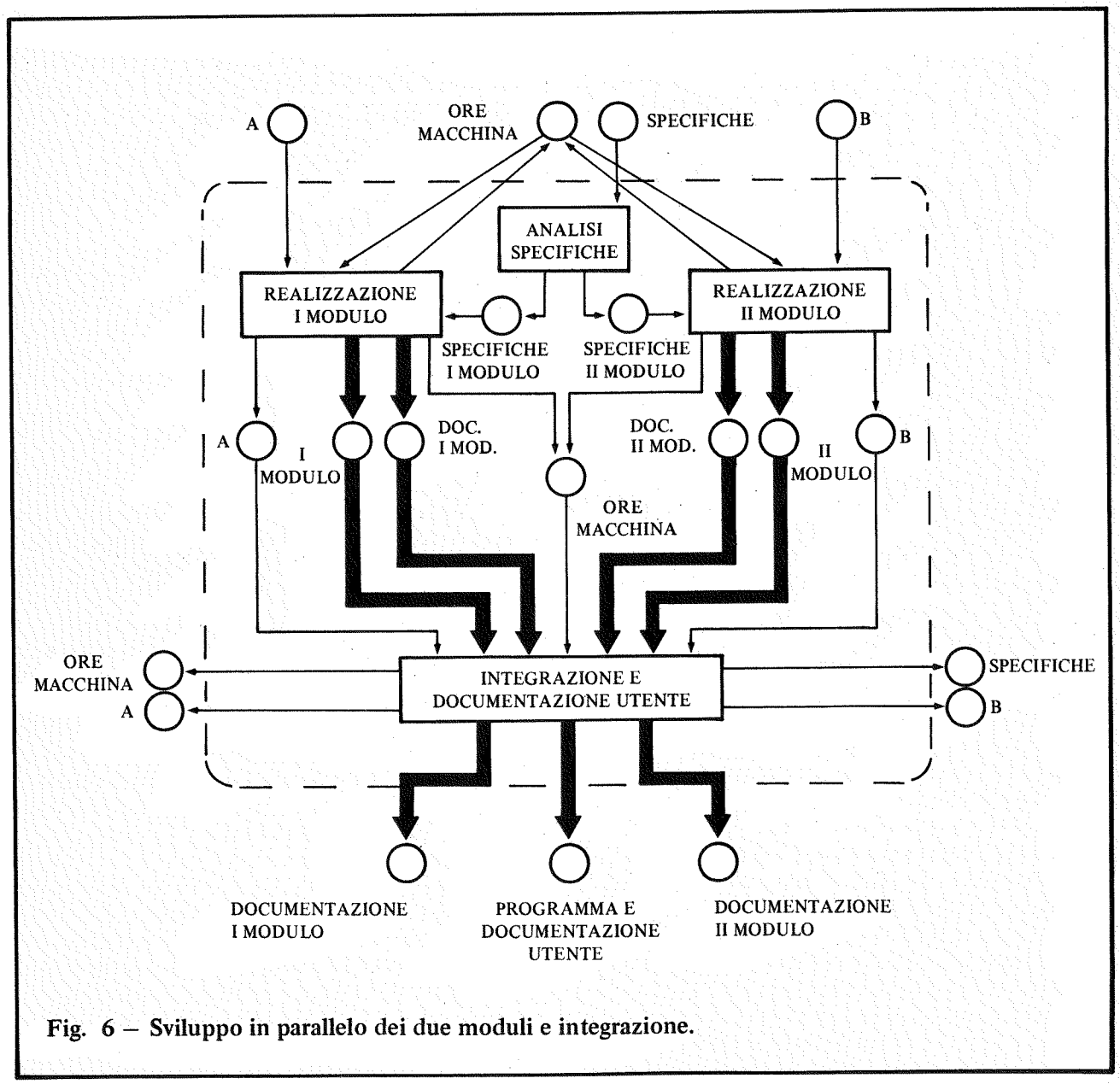



**Fig. 6 – Sviluppo in parallelo dei due moduli e integrazione.**

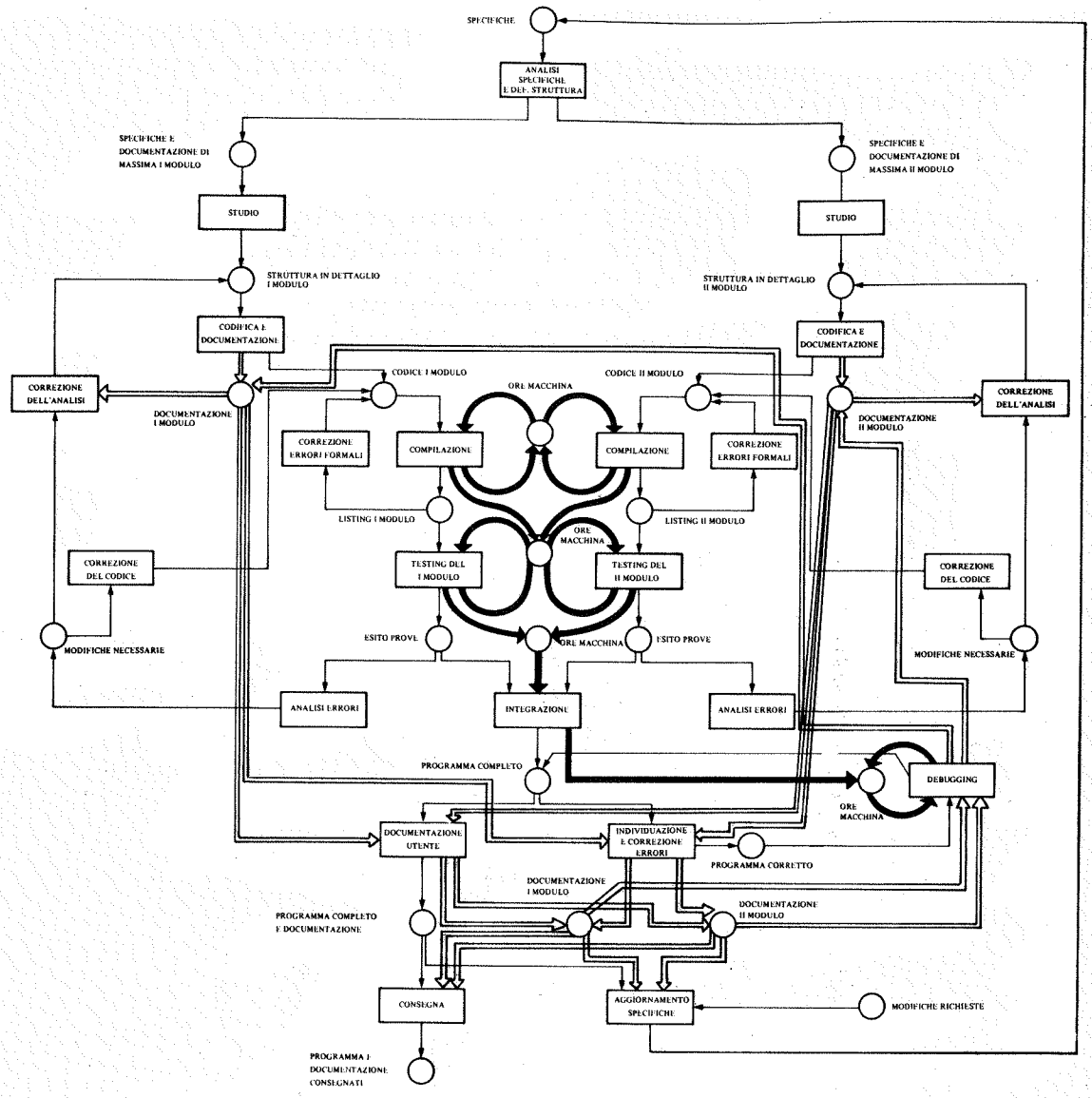



**Fig. 7**

do al decisore quando e su che cosa intervenire, fornendogli nel contempo le informazioni necessarie e uno schema delle possibili conseguenze delle sue azioni.

Una considerazione non irrilevante concerne la possibilità che la guida alla gestione, messa a disposizione dalle Reti di Petri, possa anche essere considerata come una traccia per la scelta delle informazioni, che potrebbero costituire la base per la messa a punto di sistemi informativi integrati, capaci di supportare la produzione del software. L'evidenza posta dalla Rete alle risorse e ai prodotti indica dove, nel corso del processo, essi hanno subito o subiranno trasformazioni, e suggerisce pertanto una possibile struttura relazionale che potrebbe essere mantenuta da un sistema automatico.

## 4. CARATTERISTICHE GENERALI DELLA RETE

Dall'analisi della Rete che schematizza l'esempio scelto, emergono due proprietà che hanno carattere del tutto generale per Reti di Petri con cui si intende modellare processi produttivi.

Innanzitutto, notiamo che la Rete è **viva**, ovvero ogni posto e ogni transizione possono venire raggiunti durante l'evoluzione della Rete che abbia come stato iniziale quello della "presenza delle specifiche"; ciò significa che il modello garantisce che non esistono attività inutili o risorse o prodotti non utilizzati.

Se isoliamo la sottorete di fig. 7 che non contiene gli archi **tratteggiati** ovvero se togliamo dal modello la risorsa "ore-macchina", la Rete che otteniamo è una Rete di Petri **sicura**, cioè ogni posto può avere al più un solo gettone durante una qualsiasi evoluzione della rete. L'importanza di questa proprietà è messa in luce dalla considerazione che in questo modo diventa prevedibile, mediante una simulazione, su quale dei possibili prodotti opera una attività e da quali risorse dipende, dandosi pertanto la garanzia che tutte le possibili sequenze di attività portano, a parità di prodotti, allo stesso risultato finale.

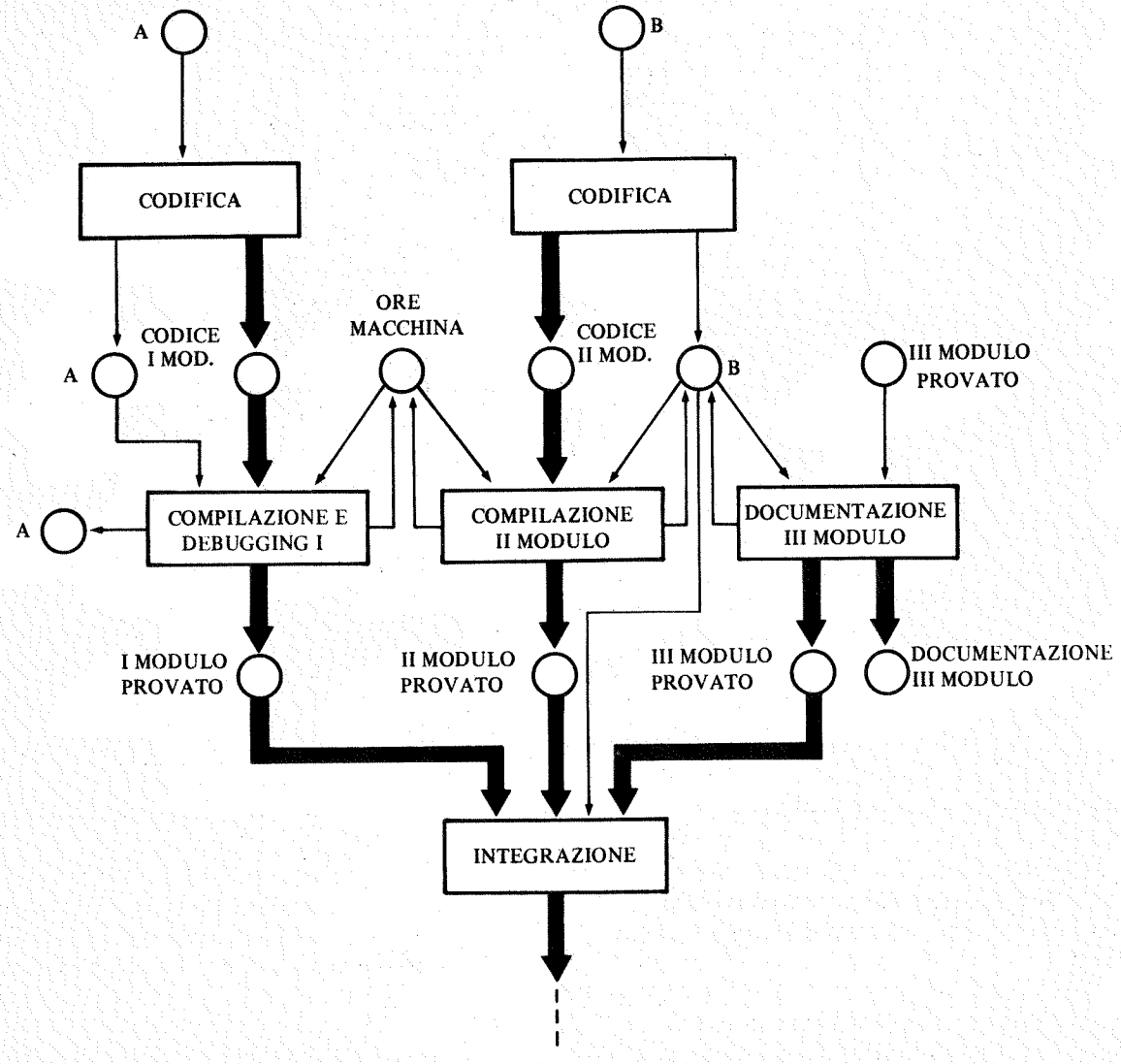



**Fig. 8 — Il problema della risoluzione dei conflitti nella gestione.**

Un'altra proprietà interessante è che l'assenza di prodotti inutili si riscontra anche alla fine dell'intero processo, allorchè l'unico posto contenente un gettone è il posto corrispondente al programma consegnato.

Infine il modello manifesta la "sopravvivenza delle specifiche" alle attività successive, ciò significa che si è inteso modellare un processo per cui il programma viene completamente sviluppato indipendentemente dall'apparire, nel corso dello sviluppo, di una richiesta di modifica delle specifiche, che viene presa in considerazione solo alla fine dell'attività di realizzazione, all'atto della decisione se il prodotto possa venire consegnato o debba essere modificato.

## 5. CONCLUSIONI

L'idea di rappresentare mediante grafi i processi produttivi non è nuova; il più significativo esempio è costituito dalle reti PERT. In questo caso la rappresentazione si basa su grafi con nodi di un solo tipo "monocromatici", in cui i nodi rappresentano momenti ben definiti nel tempo, come l'inizio o la fine di una attività, mentre gli archi mostrano l'interdipendenza tra le attività nonchè la durata dell'attività stessa. Una rete PERT può essere rappresentata come grafo "bicromatico" (associando a ogni arco una transizione, e a ogni nodo un posto), interpretandola quindi come una macchina a stati ovvero una Rete di Petri in cui ogni evento è connesso in ingresso o in uscita con un unico posto (cfr. 4).

È evidente la differente capacità espressiva dei diagrammi di PERT rispetto alle Reti di Petri. I primi infatti oltre a non permettere la rappresentazione di cicli di iterazione, danno, al contrario delle seconde, una rappresentazione fortemente sintetica di un processo, evidenziato soprattutto la dipendenza temporale tra le attività senza riuscire a mettere in luce la dipendenza dalle risorse e i conflitti che sorgono nel corso del processo. In fig. 9 abbiamo rappresentato un diagramma PERT che modella il processo produttivo presentato nell'esempio; si può qui facilmente notare come proprio le proprietà che abbiamo segnalate nel paragrafo precedente non vengono soddisfatte. Non è possibile verificare infatti l'assenza di prodotti inutili, inoltre è rappresentata una sola possibile sequenza di attività, e infine non è dato alcun rilievo al criterio di sopravvivenza delle specifiche. È innegabile d'altra parte l'utilità gestionale di tale diagramma

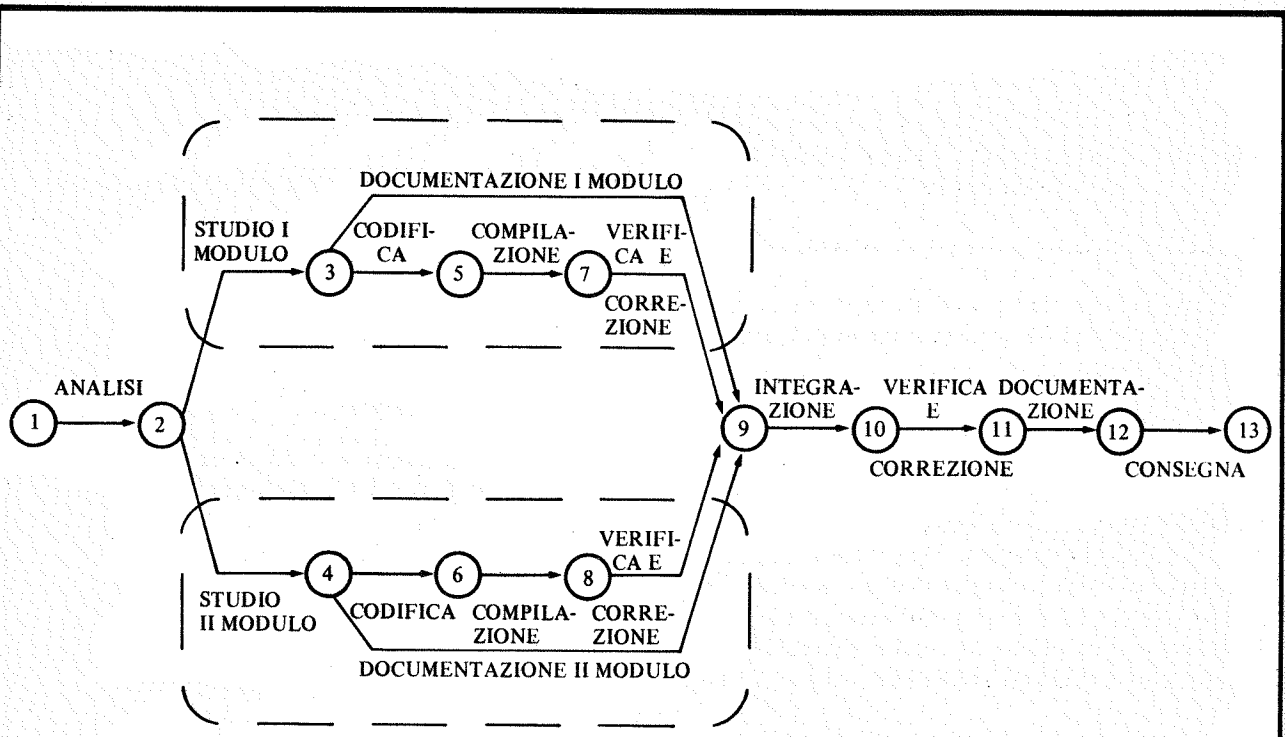



**Fig. 9 – Il PERT dell'esempio analizzato.**

per la determinazione dei tempi di produzione, che è tuttora, viceversa, resa forse più complessa dalle Reti di Petri: solo algoritmi per sistemi automatici possono aiutare a interpretare diagrammi complicati come quello presentato in fig. 7 (cfr. 3).

M.H. Hack ha introdotto per primo l'idea di utilizzare le Reti di Petri per la descrizione di processi produttivi (2). Dal punto di vista formale il suo approccio differisce dal presente nella limitazione che Hack pone ai propri schemi: egli infatti ammette soltanto Reti a libera scelta. Queste sono Reti per le quali una risorsa non può essere condivisa da più attività; ad esempio sono vietate situazioni come quella riportata in fig. 10.

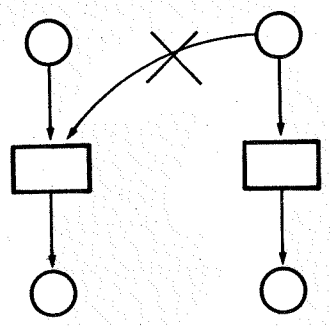


**Fig. 10 – L'arco segnato è vietato nelle reti a libera scelta.**

Questa limitazione rende difficile rappresentare i casi di risorse condivise, come ad esempio la risorsa ore-macchina.

A conclusione vogliamo riportare una descrizione (cfr. fig. 11) in linguaggio naturale del processo scelto come esempio. Si è adottata una scrittura 'indentata' per mettere in evidenza i rapporti gerarchici tra gruppi di attività. Ogni attività (semplice o composta di altre sottoattività) è preceduta da un punto (.); inoltre si sono **sottolineate** alcune parole chiave che denotano situazioni tipiche di un processo, come **parallelo, alternativa** ecc.

Questo esempio non costituisce una prova della possibilità di passare con facilità e con regole precise da una rappresentazione in linguaggio naturale a una in linguaggio formale o viceversa. È stata proposta per aiutare il lettore a cogliere i differenti poteri espressivi dei linguaggi che si possono adottare per descrivere processi produttivi. D'altra parte è evidente come la descrizione proposta sia più affine alla rappresentazione mediante rete PERT (fig. 9) del processo che a quella fornita con Rete di Petri (fig. 7). Successive ricerche saranno volta a legare la rappresentazione formale con Reti di Petri a quella in linguaggio naturale, nell'ottica di realizzare un ponte tra rappresentazioni suscettibili di elaborazione automatica e rappresentazione dei risultati nei modi piú affini alla comunicazione umana.

. A e B non appena pronte le specifiche del programma ne iniziano la realizzazione che consiste in:

  . analisi delle specifiche e definizione della struttura del programma in due moduli;
  . sviluppo **in parallelo** di due moduli che consiste in:
    . studio di ciascun modulo;
    . sviluppo **in parallelo** di:
      . documentazione di ciascun modulo;
      . codifica; compilazione; verifica e correzione del codice
  . integrazione dei due moduli;
  . verifica e correzione del programma completo;
  . sviluppo della documentazione utente;

. Appena la realizzazione sarà terminata, il programma e la relativa documentazione saranno disponibili;

. Il programma e la documentazione, appena disponibili, dovranno essere **alternativamente**:

  . consegnati;
  . **oppure** aggiornati se sono richieste modifiche delle specifiche iniziali del programma e quindi consegnati.

**Fig. 11 — Una descrizione in linguaggio naturale del processo di sviluppo dell'esempio.**

## 6. RIFERIMENTI

[ 1 ] Holt A.W., Meldman J.A., PETRI NETS AND LEGAL SYSTEMS, Jurimetrics Journal, 1965

[ 2 ] Hack M., ANALYSIS OF PRODUCTION SCHEMATA BY PETRI NETS, MIT Project MAC TR-94, Feb. 1972

[ 3 ] Ramchandani C., ANALYSIS OF ASYNCHRONOUS CONCURRENT SYSTEMS BY PETRI NETS, MIT Project MAC TR-120, Feb. 1974

[ 4 ] Peterson J.L., PETRI NETS, Computing Surveys, vol. 9, n. 3, 1977

# BOOLEAN CONTROL OF PROGRAMMING :
## come aumentare il grado di affidabilità di procedure a scarsa controllabilità intermedia.

G. Ciucci*, A. Guarnieri°, L. Quartapelle*

**Riassunto**

Un problema comune alla gran parte della produzione di software è il controllo step-by-step di una procedura al fine di individuare eventuali errori, logici o di programmazione, ed assicurare, così, un alto grado di affidabilità ai risultati. Un'analisi dell'esperienza compiuta nella realizzazione di CRYSALIS (1) ha permesso di individuare una interessante soluzione a tale problema.
Tale soluzione si basa su una tecnica di programmazione che, senza alterare la leggibilità del programma, prevede l'inserimento nel flusso logico della procedura di costrutti che operano un controllo "run time" della correttezza logica della programmazione.

**Abstract**

This note describes a programming technique which allows a step-by-step control of intermediate results of a procedure.
The technique, called "Boolean Control of Programming", has been experienced in the development of CRYSALIS (Cristal Accurate Levels by Informatic Structures), a program in which only the range of expected results was known in advance.

L'esigenza di garantire un alto grado di affidabilità ad una procedura software ha assunto particolare importanza nella realizzazione di CRYSALIS, un programma di calcolo fisico, poichè, a priori, era noto solo il range di credibilità fisica entro cui sarebbero dovuti cadere i risultati.

La soluzione adottata, detta «Boolean Control of Programming», presenta caratteristiche di efficienza e di modularità, tali da renderne utile l'uso anche al di fuori di questa specifica applicazione. Questa tecnica è stata impiegata con successo anche nella realizzazione di un modulo di sistema operativo che presiede alla gestione dei terminali.

---

(1) Il programma CRYSALIS, Crystal Accurate Levels by Informatic Structures, esegue il calcolo dei livelli energetici degli elettroni nei cristalli ortorombici della famiglia IV-VI; esso è stato progettato e realizzato dagli estensori del presente articolo presso l'Istituto di Fisica Sperimentale del Politecnico di Milano.

* Istituto di Fisica Sperimentale del Politecnico di Milano
° HISI, Pregnana Milanese

Nel prosieguo viene descritto il processo logico che ci ha portati ad adottare una tale soluzione, in modo da chiarire sia che cosa si chiede ad una struttura diagnostica per elevare il grado di affidabilità del prodotto, sia come tutto ciò sia stato ottenuto.

È noto che la sola analisi dei dati finali di una procedura può evidenziare la presenza di errori, logici o di programmazione, non garantirne l'assenza. Vi sono, infatti, errori che producono dei risultati palesemente errati e che sono, di conseguenza, immediatamente rilevabili; rimane, però, la possibilità che dati finali, prodotti da un software scorretto, siano casualmente compresi nel range di attendibilità. Questa probabilità aumenta quando il range è vasto oppure ha limiti non ben definiti.

Il grado di affidabilità di una procedura può, quindi, essere aumentato solo da un controllo step by step della correttezza del software.

Un metodo largamente impiegato è la stampa dei valori che alcune variabili assumono durante l'esecuzione: un tal modo di procedere permette di verificare la correttezza dei dati in uscita da ogni singolo step. È evidente che i tempi di messa a punto di una procedura aumentano in misura proporzionale al numero di valori da controllare, e ciò provoca una pratica inattuabilità di questo metodo nelle applicazioni che elaborano contemporaneamente un gran numero di dati: nel CRYSALIS vengono trattate matrici che arrivano a dimensioni 150 x 150, sovente come risultato di riduzioni da matrici di dimensioni molto maggiori.

Per evitare che i tempi di messa a punto di una procedura si dilatino a dismisura è, quindi, necessario prevedere, sin dalla fase di analisi, l'inserimento nel flusso logico di costrutti che verifichino automaticamente la correttezza del software.

Tali costrutti consentono di operare i controlli con tempi di ciclo di CPU e non di interazione uomo-calcolatore da tavolo. L'insieme di tali costrutti costituisce la struttura diagnostica del prodotto.

Le procedure di grosse dimensioni sono, usualmente, frutto della cooperazione di più program-

matori, con tutti i problemi di circolazione dell'informazione all'interno di un gruppo di lavoro che ne derivano; questo fatto, unitamente alle considerazioni precedenti, ci porta a concludere che una struttura diagnostica efficiente deve rispondere ai due seguenti requisiti:

- evidenziare l'eventuale presenza di errori e facilitare l'individuazione del punto esatto in cui questi errori si producono;

- conservare modularità alla procedura al fine di non ostacolare né il lavoro di gruppo né la successiva manutenzione del prodotto.

Si consideri un flusso logico simile a quello riportato in figura 1.
Ogni step del flusso è individuato da un set di dati in ingresso, un set di dati in uscita e da una o più relazioni, dette relazioni principali, che permettono di passare dai dati in ingresso a quelli in uscita.
Nella fase di programmazione e codifica, alla struttura semantica pensata dal progettista viene fatta corrispondere una struttura semantica software attuale. I dati in uscita da uno step software sono detti plausibili se non sono in contraddizione con la struttura semantica pensata.

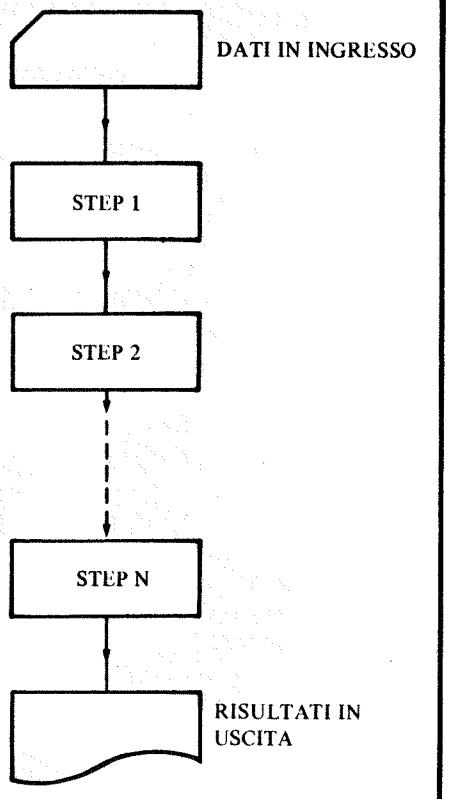



Fig. 1

Dal fatto che i dati in uscita non possono, evidentemente, contraddire la logica software che li genera, segue che condizione sufficiente affinchè i dati attuali siano plausibili è che la struttura semantica pensata sia correttamente rispecchiata dal software dello step. In altre parole, non deve avvenire che si sia pensata una cosa e poi il programma ne faccia un'altra.

La proposizione inversa afferma che dei dati plausibili sono condizione necessaria, questa volta non sufficiente, alla correttezza del software.

Il concetto di plausibilità ha, quindi, permesso di individuare un criterio di necessità per la correttezza del software.

Per effettuare dei test di plausibilità occorre individuare altre relazioni, oltre a quelle principali, fra i dati in output di uno step.
Tali relazioni sono ricavate dalla logica del particolare progetto, e vengono dette ridondanti in quanto non strettamente necessarie al processo che dai dati in ingresso porta ai dati in uscita. Il grado di plausibilità dei risultati sarà tanto maggiore quanto più restrittive saranno le relazioni ridondanti usate per i test: esse devono essere restrittive almeno quanto le relazioni principali, altrimenti i test non avrebbero alcun significato.

Nel CRYSALIS, ad esempio, alcune di queste relazioni ridondanti sono suggerite da considerazioni sulla simmetria del cristallo in esame. Una risoluzione matematicamente ineccepibile, partendo da dati simmetrici, giunge necessariamente a dei risultati che rispettano la geometria del problema; ne segue che dei test di simmetria implicano un controllo sulla correttezza matematica dei vari step ma possono svelare altre inesattezze. Nel caso del CRYSALIS, una relazione dettata da considerazioni di simmetria è, dunque, più restrittiva di relazioni ridondanti puramente aritmetiche.

Affinchè l'introduzione dei test di plausibilità nel flusso logico di un programma costituisca una struttura diagnostica rispondente ai requisiti enunciati sopra, devono essere soddisfatte le seguenti due condizioni:

- quando un test di plausibilità dà esito negativo si deve interrompere immediatamente l'esecuzione

del programma, in modo da poter individuare il punto esatto in cui si è prodotto l'errore ed evitare che errori successivi si compensino;

- l'introduzione deve avvenire senza stravolgere la struttura originaria.

Per ottenere tutto ciò, si è definita una variabile booleana, comune a tutta la procedura, che assume valore TRUE se i dati sono plausibili, FALSE altrimenti. A tale variabile è stato dato il nome BCP (Boolean Control of Programming).

Invariante comune all'esecuzione di tutto il programma è il valore TRUE della variabile BCP.
I test su BCP sono introdotti secondo una struttura «guard close», come riportato in figura 2.

Adottando una tale struttura uno step viene eseguito se, e solo se, i dati in uscita dallo step precedente

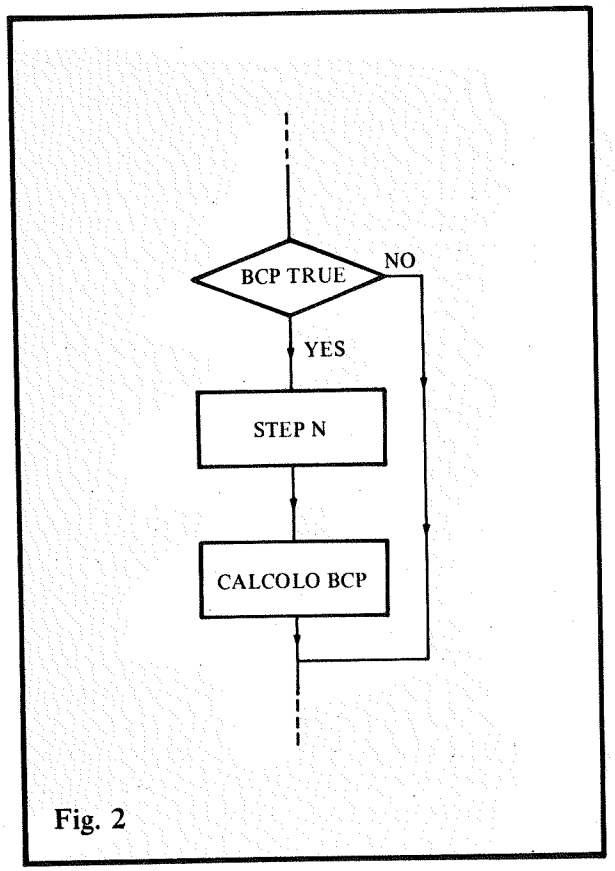



Fig. 2

sono plausibili; quando la variabile BCP assume il valore FALSE, per un errore che si è prodotto all'interno dello step, nessuno degli step successivi viene eseguito: si ha, quindi, un'interruzione nell'esecuzione del programma. Notiamo che questa interruzione avviene senza dover introdurre dei salti fuori dallo step.

La struttura «guard close» di figura 2 è ripetitiva (fig. 3) : essa può essere usata per controlli a livelli sempre più profondi, sia all'interno dello step sia in subroutine richiamate, conservando sempre la caratteristica essenziale di interrompere l'esecuzione del programma non appena un test sulla variabile BCP dia esito negativo, e questo indipendentemente dalla posizione del test. Un eventuale dump riporta la situazione di memoria dello step che contiene un errore.

La leggibilità del programma è conservata poichè si evita sia il proliferare dei salti ad errore sia la necessità di subroutine con più ritorni (il ritorno normale ed almeno un ritorno per errore).

Il fatto di poter conservare la modularità del programma anche in fase diagnostica ripaga ampiamente sia del tempo impiegato ad escogitare i test di plausibilità, sia del lieve aumento nel tempo di esecuzione. Una simile organizzazione consente il testing automatico di ciascun modulo o sottomodulo, realizzato da un diverso estensore in un lavoro a più mani, oppure a fronte di eventuali modifiche, dovute a nuove esigenze o a normale manutenzione.

In applicazioni commerciali che necessitano di un'altissima affidabilità, la struttura BCP permette di «schermare» ciascuno step da errori, di qualsiasi natura, che si siano prodotti in step precedenti; ad esempio, la fase particolarmente delicata di aggiornamento di un archivio può avvenire solo se tutti i precedenti passi sono esenti da errori.

Un ultimo, non trascurabile, vantaggio della tecnica BCP è che l'individuazione dei test di plausibilità, costringendo ad approfondire la fase di analisi, diminuisce la probabilità che un progetto parta con un'analisi incompleta.

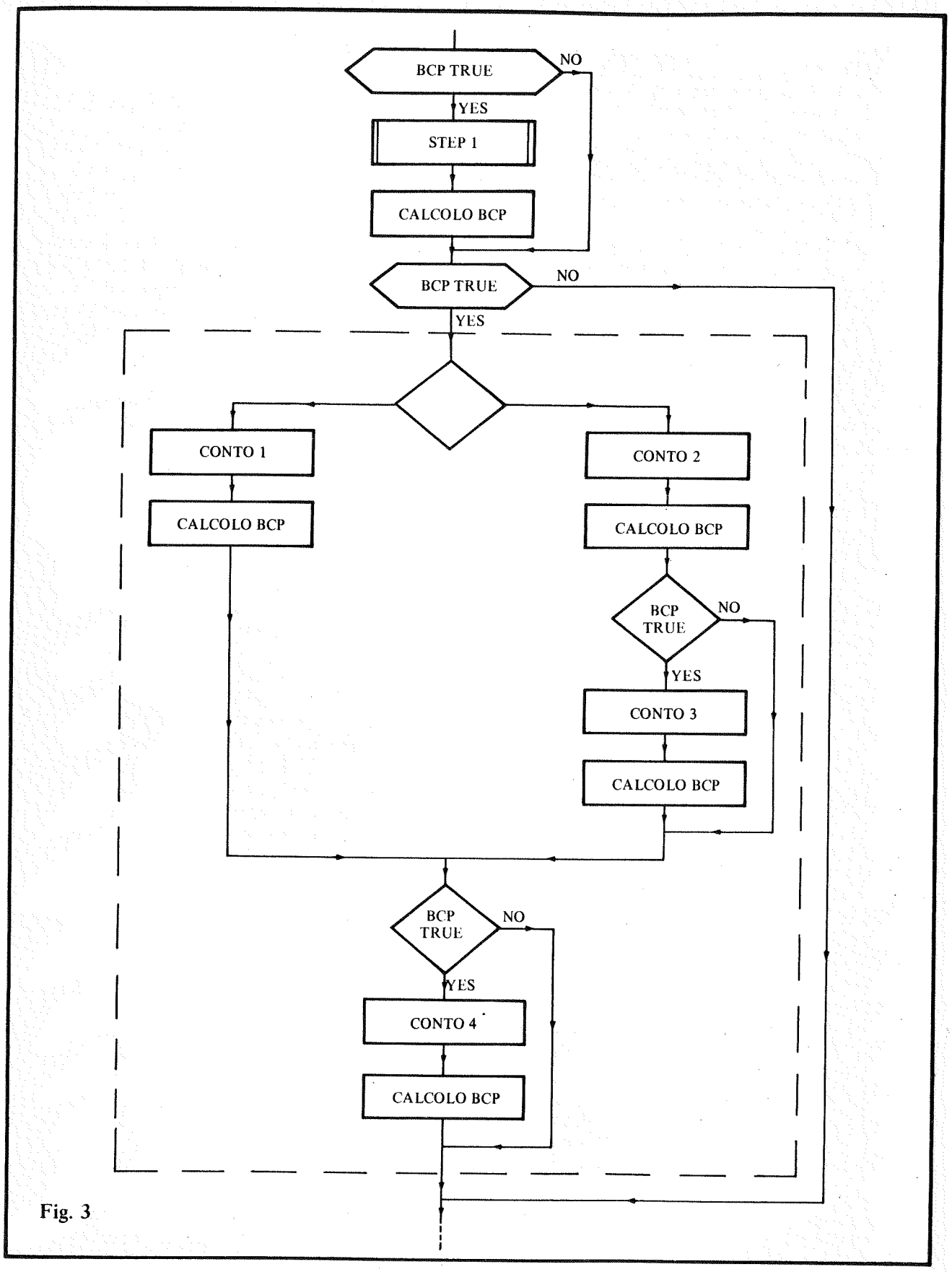

BCP TRUE
NO
YES
STEP 1
CALCOLO BCP
BCP TRUE
NO
YES
CONTO 1
CALCOLO BCP
CONTO 2
CALCOLO BCP
BCP TRUE
NO
YES
CONTO 3
CALCOLO BCP
BCP TRUE
NO
YES
CONTO 4
CALCOLO BCP

Fig. 3

# APPROCCI ALLA COMUNICAZIONE DI ATTIVITÀ CONCORRENTI IN SISTEMI DI ELABORAZIONE DISTRIBUITI

G. Perego*

Istituto di Scienze dell'Informazione dell'Università di Pisa

**Riassunto**

I problemi della comunicazione tra attività concorrenti in sistemi distribuiti senza memoria comune stanno assumendo sempre maggiore importanza.
In questo lavoro vengono indicati alcuni limiti del concetto di monitor e dei linguaggi sviluppati intorno ad esso per applicazioni su suddetti sistemi.
Due recenti proposte in questo campo sono poi presentate, confrontate e discusse, mostrando come un approccio basato su distribuzione delle funzionalità e comunicazione come concetti primitivi permetta di risolvere molti dei problemi legati allo sviluppo di costrutti di programmazione quali monitors, classi, coroutines etc.

**Abstract**

This paper discusses some limitations of the monitor concept, and of languages based on it, when it is used in distributed systems without common memory.
Two recent proposals are then reviewed and discussed with examples: the "Communicating Sequential Processes" of C.A.R. Hoare, and the "Distributed Processes" of Brinch Hansen.

## 1. INTRODUZIONE

I linguaggi ad alto livello per la programmazione concorrente più noti, come il Concurrent Pascal [ 2 ] e il Modula [ 13 ], così come sono concepiti ed implementati, sono strumenti adatti al progetto di sistemi multiprogrammati su singolo processor, o al massimo su multiprocessor con memoria comune. Il Modula, addirittura, è fortemente orientato verso implementazioni su uniprocessor, mentre per il Concurrent Pascal sarebbe probabilmente possibile l'adattamento anche a sistemi ove ogni processor può accedere direttamente soltanto a una propria area di memoria privata, imponendo però alcuni vincoli sui meccanismi di passaggio dei parametri, ad esempio nella direzione indicata da Brinch Hansen in [ 4 ].

*Lavoro svolto nell'ambito della collaborazione tra Honeywell Information Systems Italia e Istituto di Scienze dell'Informazione dell'Università di Pisa.

Tuttavia, linguaggi di questo tipo rendono difficile sfruttare i vantaggi offerti da questi tipi di sistemi, e sembrano quindi poco adatti alla loro programmazione. Parallelamente assumono sempre più rilevanza le problematiche connesse a sistemi distribuiti ove i processi comunicano mediante scambi di messaggi, sia per gli sviluppi tecnologici, che spingono verso una sempre maggiore distribuzione del controllo, che per l'allargarsi dell'area di applicazione della programmazione concorrente.

Recentemente sono state avanzate due proposte [ 4 ], [ 8 ], particolarmente attraenti per la loro semplicità, e molto interessanti dal punto di vista concettuale, in cui processi paralleli con diritti di accesso solo a variabili locali e comunicazione come interazione con l'ambiente esterno sono visti come concetti primitivi.

Cercheremo qui di indicare i limiti di linguaggi come Concurrent Pascal e Modula, e del concetto di monitor [ 7 ] intorno a cui sono stati costruiti, per la programmazione dei sistemi distribuiti con memoria locale, e di far vedere come concetti come monitors, classi, coroutines ecc. sono sostanzialmente uinificabili in un approccio basato su processi concorrenti, che comunicano e si sincronizzano mediante scambi di messaggi.

## 2. I MONITORS ED ALCUNI DEI LORO LIMITI

Non scenderemo qui nei dettagli relativi all'implementazione a all'uso del concetto di monitor rimandando per questo il lettore alla letteratura riferita.

In breve, i monitors consistono di strutture dati ripartite, sintatticamente associate a un insieme preciso di operazione (procedure), che sono le uniche permesse su tali strutture dati. I processi, quando devono interagire fra loro, per utilizzare tali strutture dati o anche soltanto per sincronizzare le proprie operazioni, chiamano le procedure dei monitors. Attraverso un nucleo run-time viene imple-

mentata sui monitors la mutua esclusione tra i processi concorrenti, e viene eseguito lo smistamento dei processi stessi.

Il nucleo consiste essenzialmente delle strutture dati che rappresentano i processi, delle procedure che manipolano tali strutture dati e di quelle che implementano gli ingressi e le uscite dal monitor e le primitive di sincronizzazione che all'interno del monitor vengono usate. La struttura di questo nucleo dipende strettamente sia dalle caratteristiche fisiche del sistema per cui esso funziona, sia dal tipo di sistema operativo che si vuole costruire sopra di esso. In [ 2,12,13 ] è possibile trovare la descrizione dettagliata di tre diverse implementazioni di nucleo run-time, ciascuna disegnata in base a precise esigenze. In ogni caso, le dimensioni di questa parte del sistema possono essere assai limitate, ed essendo essa l'unica che deve essere programmata direttamente in linguaggio a basso livello, ciò costituisce un grosso vantaggio dal punto di vista dell'affidabilità e del costo.

Attorno al concetto di monitor sono stati recentemente sviluppati dei linguaggi ad alto livello per la programmazione concorrente, quali Concurrent Pascal e Modula, che rappresentano un passo avanti di grande rilevanza per quanto riguarda il problema della riduzione dei costi di sviluppo e l'aumento dell'affidabilità dei sistemi operativi, dei sistemi per applicazioni real-time, ed in genere del software concorrente.

Tuttavia questi linguaggi, ed il concetto di monitor stesso, non soddisfano le necessità di altri tipi di applicazioni quali quelle real-time controllate da reti di microcomputer con memoria distribuita. Tali applicazioni, tra le più difficili per i vincoli critici di tempo ed affidabilità, si vanno sempre più diffondendo parallelamente allo sviluppo tecnologico nel campo dei microprocessori e dei circuiti LSI e VLSI.

I limiti di tali linguaggi sono sia di tipo concettuale che di tipo implementativo.
Innanzitutto, il monitor è una componente di programma passiva, viene cioè chiamato dai processi che vogliono utilizzare le sue strutture dati ripartite, e le operazioni (procedure) su questi dati vengono eseguite sotto il controllo diretto dei processi chiamanti. Ciò implica, per una implementazione diretta, che:

1 ) le strutture dati risiedono in una memoria comune, cui tutti i processi devono poter accedere, indipendentemente dal processor su cui sono in quel momento in esecuzione. In questo modo la memoria comune diviene una pesante strozzatura per il sistema, sia che essa sia fisicamente un'area di memoria cui tutti i processor possono accedere direttamente attraverso una rete di arbitrazione, sia che sia fisicamente distribuita come parte delle memorie locali dei vari processors, direttamente indirizzabile da ciascuno di essi (caso questo anche peggiore).

2) . Richiesta ed esecuzione dei servizi gestiti dai monitors sono affidate entrambe al controllo diretto del processo che ne abbisogna, in qualsiasi situazione. Questo può ridurre, in alcune situazioni, il grado di parallelismo effettivo rispetto a quello possibile [ 9 ].

È certo possibile adattare i monitors ai vincoli imposti dalla presenza di memoria locale ad un unico processor, su sistemi distribuiti, tuttavia per questi sistemi è preferibile un approccio diverso. Infatti utilizzando parallelismo e comunicazione come concetti primitivi, cosa estremamente naturale in questo ambito, è possibile sia implementare molti dei concetti della programmazione (coroutines, pipelining, monitors, procedure, ecc...), sia ripulire in modo sostanziale i linguaggi dal punto di vista concettuale.

Le proposte che ora prenderemo in esame rappresentano un primo sforzo in questa direzione.

## 3. I PROCESSI SEQUENZIALI COMUNICANTI DI HOARE

Questo linguaggio [ 8 ] propone l'unificazione di diversi metodi di sincronizzazione fra processi, e di diversi concetti di programmazione, attraverso l'adozione dell'ingresso e dell'uscita come primitive di base e dei «guarded command» di Dijkstra [ 6 ] come strutture di controllo e strumenti per introdurre e controllare il nondeterminismo.

Usando i «Processi Sequenziali Comunicanti» è possibile specificare un sistema ed il suo funzionamento attraverso una configurazione fissa di processi paralleli, ciascuno con una precisa sottofunzione, tra loro comunicanti in modo strettamente sincronizzato mediante comandi di ingresso e di uscita.

Non riporteremo la descrizione completa del linguaggio, che può essere reperita nell'articolo di Hoare riferito, ma soltanto alcune delle sue caratteristiche più importanti ed inoltre alcuni esempi di

programma per darne, in modo informale, un'idea più precisa.

### 3.1 - Il linguaggio

Il parallelismo tra i processi viene espresso mediante un comando la cui sintassi è:

« parallel command » : : =
[ « process » { || « process » } ]

(le parentesi graffe indicano zero o più ripetizioni di quanto in esse contenuto).

I processi di un comando parallelo devono essere disgiunti, vale a dire che le intersezioni tra i loro domini e ranghi devono essere a due a due vuote.

Un comando parallelo provoca l'inizio simultaneo dell'esecuzione di tutti i processi che lo costituiscono. Il comando termina con successo solo quando essi sono tutti terminati con successo.

Ogni processo è formato da una etichetta (che può essere vuota) e da una lista di comandi.
Attraverso la lista di comandi si specifica la sequenzializzazione dei comandi contenuti.
L'etichetta di un processo può avere degli indici, associati ad uno o più ranghi di variabilità: in questo caso il processo denota una serie di processi, ciascuno con la stessa etichetta e lista di comandi, ma con una diversa combinazione di valori sostituiti alle variabili legate. In questo modo si possono indicare matrici di processi.

Ad esempio,

Y ( j = 1 ..n) : : Command List

denota

Y ( 1 ) : : Command $List_1$ ||
Y ( 2 ) : : Command $List_2$ || ...
Y ( n ) : : Command $List_n$

Nessuna assunzione viene fatta riguardo alla velocità relativa di esecuzione dei processi di un comando parallelo, cioè i processi contenuti possono essere eseguiti in modo asincrono, e con qualsiasi sequenza di iniziazioni e terminazioni. La possibilità di sincronizzazione verrà poi introdotta come vedremo.

Il linguaggio contiene l'usuale statement di assegnamento, e permette l'assegnamento parallelo
( x,y ) : = ( y,x ) .
È possibile utilizzare costruttori ed eseguire assegnamenti del tipo x : = Cons ( left, right ), etc.

La comunicazione attraverso i processi operanti concorrentemente è ottenuta mediante comandi di input e di output :

« input command » : : = « source » ?
« target variable »
« output command » : : = « destination » !
« expression »
« source » : : = « process name »
« destination » : : = « process name »

I processi indicati come destinazione possono essere processi hardware, quali ad esempio lettori di schede o stampanti, oppure scelti tra i processi costituenti un comando parallelo.

La semantica di questi comandi è la seguente.
La comunicazione fra due processi avviene se:

1 ) Un comando di input in un processo specifica come sua sorgente il nome dell'altro processo.

2 ) Un comando di output nell'altro processo specifica come sua destinazione il nome del primo processo.

3 ) La « target variable » del comando di input e il valore dell'espressione del comando di output hanno tipo compatibile, ed è possibile un assegnamento tra loro.

Con queste condizioni, i due comandi di input e di output corrispondono, e sono eseguiti simultaneamente. L'effetto è l'assegnamento alla « target variable » nel comando di input del valore della espressione nel comando di output. Se la sorgente di un comando di input è terminata, il comando fallisce.
Un comando di output fallisce se la sua destinazione è terminata oppure il valore dell'espressione è indefinito. La necessità di sincronizzare ingresso ed uscita impone che un processo che esegue uno di questi comandi sia ritardato fino a che il comando corrispondente non viene eseguito dall'altro. In questo modo i processi, operanti altrimenti asincronamente, possono essere sincronizzati tra loro.

Esempi :

| | | |
|---|---|---|
| ( 1 ) | Giovanni? ( cognome, età ) | da Giovanni riceviunacoppia di valori e assegnaliacognome e età. |
| ( 2 ) | Agenda! ( Rossi, 30 ) | aAgendaspedisci una coppia di valori |

Se ( 1 ) e ( 2 ) corrispondono, nel senso specificato, la comunicazione avviene e a cognome è assegnato il valore Rossi, a età il valore 30.

| | | |
|---|---|---|
| ( 3 ) | Y ( j ) ? P ( ) | dal j-esimo processo della matrice Y, ricevi un segnale P ( ), rifiutando qualsiasi altro tipo di segnale. |
| ( 4 ) | sem! V ( ) | manda un segnale V( ) a sem. |

Infine, come comandi strutturati del linguaggio, sono adottati i « guarded command » di Dijkstra [ 6 ], il comando alternativo e il ripetitivo:

« repetitive command » : : =
    * « alternative command »

« alternative command » : : =
    [« guarded command »
    { [] « guarded command »} ]

« guarded command » : : =
    « guard » → « command list »|
    ( « range » { , « range »} )
    « guard » → « command list »

« guard » : : =
    « guard list »|
    « guard list »; « input command »|
    « input command »

« guard list » : : =
    « guard element » { ; « guard element »}

« guard element » : : = « boolean expression »|
    « declaration »

Un guarded command è eseguito soltanto se almeno una delle sue guardie non fallisce.

Le espressioni booleane vengono valutate, e la lista di comandi eseguita se e sclo se il valore ottenuto è «vero». Il comando di input, situato alla fine della guardia (i cui elementi vengono valutati da sinistra a destra), viene eseguito se e quando un comando di output corrispondente viene eseguito in un altro processo.

Il comando alternativo specifica l'esecuzione di uno soltanto dei guarded command che lo costituiscono. Il comando fallisce se tutte le guardie falliscono, e, nel caso che più di una termini con successo, una delle liste di comandi è arbitrariamente scelta per essere eseguita.

Il comando ripetitivo specifica iterazione, fino a quando almeno una delle guardie del comando alternativo termina con successo.
Quando tutte le guardie falliscono, il comando ripetitivo termina senza nessun effetto, è cioè equivalente allo statement nullo.

L'uso di guarded command e guardie di input permette di specificare il modo in cui processi diversi devono sincronizzare le proprie operazioni.

### 3.2. - Due esempi

Hoare, nell'articolo citato, dà una serie di esempi estremamente completa, applicando il suo linguaggio a molti dei problemi classici della letteratura. Ci limiteremo quindi, per far almeno intravedere il tipo di struttura dei programmi che si possono scrivere usandolo, a due esempi :

a) un programma che legga schede di 80 caratteri da un processo hardware LETTORE, e ne stampi il contenuto su un processo hardware STAMPANTE in righe di 125 caratteri, inserendo uno spazio bianco alla fine di ogni scheda e completando l'ultima riga con bianchi (fig. 1).
Skip indica lo statement vuoto.

b) Un programma che ottimizza il movimento della testina di un disco. La strategia di ottimizzazione è quella di minimizzare i cambiamenti di direzione della testina. Questa strategia, tra l'altro, previene la starvation indefinita per qualche processo.
Come si può vedere, le restrizioni del linguaggio impongono che tutti i processi conoscano il nome del processo DISKHEAD SCHEDULER (e fin qui nulla di male) e viceversa.
Esisteranno n processi, (process ( 1 ) . . process ( n ) ), che useranno il disco nel modo seguente :

. . . ; dhs! cylnumber; dhs! REQUEST;
. . . accedi al disco . . . ; dhs! RELEASE . .

Il processo viene definito in fig. 2.

```
UNPACK : : scheda : ( 1 . .80 ) character;
           * [ LETTORE? scheda → i : integer; i : = 0
                                 * [ i < 80 → PACK! scheda ( i ); i : = i + 1 ]
                                 PACK! "spazio"
           ]||

PACK : : linea : ( 1 . .125 ) character; i : integer ; i : = 0;
         * [ c : character; UNPACK? c → linea ( i ) : = c;
                                 [ i < 124 → i : = i + 1;
                                 [] i = 124 → STAMPANTE! linea; i : = 0
                                  ]
         ]
         i = 0 → skip
         [] i > 0 → * [ i < 124 → linea ( i ) : = " spazio "; i : = i + 1 ];
                    STAMPANTE! linea
         ]
]
```

**Fig. 1**

### 3.3. - Considerazioni

È interessante il fatto che le operazioni di comunicazione tra processi, che nei linguaggi tradizionali per la multiprogrammazione sono eseguite mediante stutture dati (e quindi memoria) comuni, sono qui viste come operazioni di input/output. Inoltre, le relazioni tra i processi sono perfettamente simmetriche. Questa visione, che è possibile trovare in alcuni recenti lavori sulla semantica dei processi concorrenti (seppure più generalizzata) [ 10,11 ], è estremamente elegante ed ha conseguenze importanti sia sul tipo di struttura che è possibile dare al software scritto usando questo linguaggio, sia sulla architettura dei sistemi che devono supportare l'implementazione del linguaggio stesso.

Infatti, la necessità di conoscere il nome del processo con cui si vuole comunicare, e viceversa, può creare dei problemi per programmi concorrenti a struttura gerarchica, in cui un processo servente dovrebbe, in linea di principio, non conoscere l'identità del proprio master.
Inoltre sembra naturale pensare all'uso di questo linguaggio per sistemi lascamente connessi, ove non esista uno spazio di memoria comune direttamente indirizzabile dai processi. L'implementazione più immediata cui viene da pensare è quella attraverso un anello.

Comunque, sebbene nella sua struttura attuale il linguaggio sia piuttosto rigido, riteniamo che sia possibile modificarlo, pur mantenendone le caratteristiche fondamentali, per adattarlo alle esigenze dei sistemi multimicro a memoria distribuita e delle reti locali.

Tra l'altro, la soluzione di problemi come deadlock e starvation non è affrontata nel linguaggio, e deve essere risolta dall'implementazione.
La struttura statica dei sistemi di processi che è possibile specificare attraverso il linguaggio dovrebbe rendere possibile il rilevamento in fase di compilazione di possibilità di deadlock, mentre della starvation è necessario occuparsi durante l'implementazione del meccanismo che gestisce le comunicazioni tra i processi.

La scelta di permettere soltanto comandi di input nelle guardie non è giustificata da Hoare, che anzi afferma l'utilità di introdurre simmetria anche in questo.
Anche questa, a nostro parere, è una questione che deve essere risolta al momento dell'implementazione delle primitive di comunicazione del linguaggio.

```
. . . ||dhs : : direction : boolean; reg : (1 . .n) boolean;
              req : = false; direction : = true;
              headpos : integer; headreq : (1 . . n) integer;
              headpos : = 0;
              *[ (i : 1 . . n) req = false; process (i)? headreq (i);
                          req (i) : = true;
                          [ headreq (i) = headpos → SKIP
                          []headreq (i) ≠ headpos → headpos : = headreq (i);
                                                    diskhead! headpos
                          ]; process (i) ? REQUEST
              [] (i : 1 . . n) req ≠ false; process (i)? headreq (i) → req (i) : = true
              [] (i : 1 . . n) process (i)? RELEASE →
                          [ req ≠ false → j : integer; j : = 0;
                                          x : (1 . . n) integer;
                                          [ direction → k : integer; k : = 1;
                                                        * [ k ≤ n; headreq (k) ≥ headpos; req (k) →
                                                                x (k) : = headreq (k); k : = k + 1
                                                        [] k ≤ n; headreq (k) < headpos; req (k) →
                                                                x (k) : = 2 * cylnumber-headreq (k);
                                                                k : = k + 1
                                                        [] k ≤ n; ¬req (k) →
                                                                x (k) : = 2 * cylnumber; k : = k + 1
                                                        ];
                                                        min : integer;
                                                        min : = cylnumber; k : = 1
                                                        * [ k ≤ n; x (k) ≤ min →
                                                                j : = k; min : = x (k) ; k : = k + 1
                                                        [] k ≤ n; x (k) ≥ min → k : = k + 1
                                                        ];
                                                        [ x (j) > cylnumber →
                                                                direction : = ¬direction
                                                        [] x (j) ≤ cylnumber → SKIP
                                                        ]
                                          [] ¬direction → k : integer; k : = 1
                                                        *[ k ≤ n;headreq (k) ≤ headpos;req (k) →
                                                                x (k) : = headreq (k);k : = k + 1
                                                        [] k ≤ n;headreq (k) > headpos;req (k) →
                                                           x (k) : = headreq(k) - cylnumber;
                                                           k : = k + 1
                                                        [] k≤n;¬req (k) → x (k) : = 0; k : = k + 1
                                                        ];
                                                        k : = 1; max : integer; max : = 0;
                                                        *[ k ≤ n; x (k) > max → j : = k;
                                                        max: = x (k); k : = k + 1
                                                        [] k ≤ n;x (k) ≤ max → k : = k + 1
                                                        ];
                                                        [ x (j) ≤ 0 → direction : = ¬direction
                                                        [] x (j) > 0 → SKIP
                                                        ]
                                          ];
                                          [ headreq (j) = headpos → SKIP
                                          [] headreq (j) ≠ headpos → headpos : = headreq (j);
                                                                     diskhead! headpos
                                          ];
                                          process (j)? REQUEST;
                                          req (j) : = false
                          [] req : = false → SKIP
                          ]
              ]
|| . . . .
```

Fig. 2

Per quanto riguarda il problema delle prove di correttezza, riteniamo che, se la struttura del linguaggio spinge verso una alta scomposizione di un sistema, e quindi riduce le dimensioni dei singoli processi, dall'altra l'estrema flessibilità delle primitive di comunicazione introduce complessità notevoli. Questo punto richiede quindi ulteriori approfondimenti.

## 4. DISTRIBUTED PROCESSES

Anche questo secondo linguaggio [ 4 ] viene proposto per applicazioni real-time controllate da reti di calcolatori con memoria distribuita.

Le caratteristiche principali del linguaggio, come le descrive Brinch Hansen, sono le seguenti:

1) Un programma consiste di un numero fisso di processi concorrenti che vengono fatti partire simultaneamente e non terminano mai.
   Ogni processo ha delle variabili proprie, le uniche cui può accedere. Non esistono variabili comuni.

2) Un processo può chiamare delle procedure comuni definite in altri processi. Queste procedure vengono eseguite quando i processi che le contengono sono in attesa su qualche condizione. Questa è la sola forma di comunicazione tra processi permessa.

3) I processi vengono sincronizzati per mezzo di statements nondeterministici chiamati "guarded regions" [ 3 ].

### 4.1. - Il Linguaggio

La struttura di un "processo distribuito" è la seguente:

**process** name
variabili proprie
procedure comuni
comando iniziale.

Un processo può eseguire due tipi di operazioni: o il comando iniziale oppure le "richieste esterne", cioè le chiamate delle procedure comuni eseguite dagli altri processi.

Il processo, quando viene attivato, esegue il suo comando iniziale, e continua fino a quando questo termina, oppure si deve attendere che qualche condizione divenga vera. Allora può essere fatta partire un'altra operazione, in risposta ad una richiesta esterna. A sua volta questa seconda operazione o terminerà o si metterà in attesa di una condizione, così permettendo di partire con un'altra operazione ancora, oppure di riprenderne una sospesa, la cui condizione è diventata vera.
Anche qualora il comando iniziale termini, il processo continuerà ad esistere e accetterà le richieste esterne.

Da notare che il passaggio di un processo da una operazione ad un'altra è controllato dal programma, e non da un clock a livello macchina.
Un processo è fermo soltanto nel caso che tutte le sue operazioni correnti siano in attesa nelle guarded regions, oppure quando ha fatto una richiesta ad un altro processo, ed attende il completamento dell'operazione richiesta.

La struttura di una procedura è la seguente:

**proc** nome (param. di ingresso # param. di uscita)
variabili locali
comando

e quella di una chiamata

**call** Q.R (espressioni, variabili)

ove Q è il nome di un processo ed R quello di una sua procedura.

Il linguaggio, per controllare il nondeterminismo, fa uso dei guarded commands, come quello di Hoare, solo con una sintassi lievemente diversa:

« alternative command » : : =
**if** « guarded command »
{| « guarded command »} **end**

« repetitive command » : : =
**do** « guarded command »
{| « guarded command »} **end**

« guarded command » : : =
« guard » : « command list »

e con la stessa semantica, ed inoltre delle guarded regions, per ritardare le operazioni, e permettere quindi la sincronizzazione tra i processi.

Le guarded regions hanno la seguente sintassi:

« when command » : : =
**when** «condition» : « command list »
{| « condition » : « command list »}
**end**

« cycle command » : : =
**cycle** « condition » : « command list »
{| « condition » : « command list »} **end**

Un processo che esegue un comando **when** attende fino a che una delle condizioni è vera ed esegue la lista di comandi corrispondente.

Il comando **cycle** specifica la ripetizione infinita del comando **when**. Come per i guarded commands, in caso di conflitto tra più possibili liste di comandi (più condizioni vere) la scelta, nondeterministica, dipenderà dall'implementazione.

I tipi di dati ammessi dal linguaggio sono interi, booleani e caratteri come tipi primitivi, ed inoltre insiemi finiti, sequenze e vettori.

Infine, il linguaggio contiene un comando **for**, di tipo enumerativo (**for** x **in** y: C **end**), e la possibilità di specificare matrici di processi (**process** name [ n ]).

```
process UNPACK:scheda : array [ 80 ] char;
                        i:integer;
do true:
   call lettore.input (scheda);
   i : = 0
   do i < 80 :
      call PACK.spedisci ( scheda (i) );
      i : = i + 1;
      |i = 80 :
      call PACK.spedisci ("spazio");
   end
end


process PACK:linea : array [ 125 ] char;
                     i:integer;
proc spedisci (c:char);
     linea (i) : = C;
     i : = i + 1;
end
i : = 0;
cycle i = 124 :
     call stampante.output (linea);
     i : = 0;
end
```

**Fig. 3**

### 4.2. - Esempi

Anche per questo secondo linguaggio daremo soltanto un paio di esempi, che ci saranno utili anche per confrontare tra loro le due proposte.

a) In fig. 3 è mostrato il programma analogo all'esempio a del par. 3.2.

Il tipo di comunicazione impedisce di avere una relazione simmetrica tra due processi.
Per ottenere questo obiettivo sarebbe stato necessario introdurre tra i due un terzo processo, con funzioni di buffer, contenente due procedure operanti sulla stessa struttura dati (è il classico buffer tra produttore e consumatore). Per ottenere una sincronizzazione stretta è necessario, invece, il programma, piuttosto inelegante, sopra riportato. Ovviamente il discorso può anche essere rovesciato: per ottenere bufferizzazione, col linguaggio di Hoare, bisogna introdurre un processo ad hoc, non naturalmente imposto dalla struttura del linguaggio. Inoltre il fatto che nella proposta di Brinch Hansen si assume che un processo non termini mai impone la trattazione esplicita di condizioni come la fine del file di lettura.

In fig. 4 riportiamo lo stesso programma, ma che si occupa anche di riempire con spazi bianchi l'ultima riga, come faceva quello dell'esempio 3.2.a.

b) Il secondo esempio è un programma che risolve il problema dei 5 filosofi di Dijkstra.

Cinque filosofi passano la loro vita pensando e mangiando. Per mangiare hanno a disposizione un tavolo rotondo, con cinque posti e cinque forchette. Ogni filosofo, per mangiare, deve usare due forchette. Allora, quando ha fame, egli va al tavolo, prende le due forchette a destra e a sinistra del suo posto se sono libere, e mangia.
Quando ha finito, depone le forchette e se ne va.

Il programma dato non elimina la possibilità che due filosofi facciano morire di fame quello in mezzo a loro, mangiando alternativamente: per evitare questo è necessario introdurre un meccanismo in cui vengono registrate le richieste arrivate e non servite dal processo « tavola », con una priorità crescente col tempo (una coda FIFO), cosa che può essere fatta, ad esempio, per mezzo di un vettore in cui le richieste vengono inserite in ordine di arrivo, se non è possibile soddisfarle immediatamente.
Il programma non tratta questo problema perchè la cosa non è rilevante dal punto di vista del linguaggio (fig. 5).

```
process UNPACK:scheda:array [ 80 ] char;
                    i:integer; esci: boolean;
do true:
   call lettore.input (scheda);
   i : = 0; esci : = false;
   do (i < P0) ∧ (esci = false):
      if (scheda (i) ≠ "eof"):
         call PACK.spedisci (scheda (i));
         i: = i + 1;
         | (scheda (i) = "eof"):
         call PACK.spedisci ("eof");
         esci: = true;
      end
      | (i = 80) ∧ (esci = false) :
      call PACK.spedisci ("spazio");
   end
end

process PACK:linea:array [ 125 ] char;
                i:integer;
proc spedisci (c:char);
if c ≠ "eof":
   linea (i): = c;
   i: = i + 1;
   | c = "eof":
   linea (i): = "eof";
   do i < 124 :
      linea (i): = "spazio";
      i: = i + 1;
   end
end
end
i: = 0;
cycle i = 124:
      call stampante.output (linea);
      i: = 0;
end
```

**Fig. 4**

### 4.3. - Considerazioni

Questo secondo linguaggio, pur avendo in comune con quello di Hoare la distribuzione del controllo, è sensibilmente diverso. Innanzitutto le comunicazioni tra i processi sono di tipo gerarchico: un processo ne chiama un altro, unilateralmente. Questo elimina la necessità della reciproca conoscenza tra i processi.

Inoltre, mentre il meccanismo di comunicazione di Hoare prevede un controllo dinamico del tipo di

```
process filosofo [ 5 ];
do true:pensa;
      call tavola.vai (this);
      mangia;
      call tavola.lascia (this);
end


process tavola:commensali:set [ 5 ] int;
proc vai (i:int);
when ([ i ⊕ 1,i ⊖ 1] ∧ commensali = [  ]) :
      commensali.include (i)
end
proc lascia (i:int);
commensali.exclude (i)
end
commensali: = [  ];
```

**Fig. 5**

variabili ed espressioni, prima che una comunicazione avvenga, qui i punti di accesso ai processi distribuiti sono dichiarati esplicitamente.

In sostanza, il linguaggio di Brinch Hansen sembra complessivamente meno elegante e conciso di quello di Hoare, e i concetti di base che esso contiene sono più numerosi e meno flessibili. Infatti, i concetti coinvolti sono quelli di processo, procedura e regione critica condizionale,mentre processi, ingresso e uscita sono sufficienti per Hoare.

Inoltre nessuna assunzione viene fatta relativamente alla terminazione dei processi.
Nell'esempio 3.2.a, ci si serviva della uscita dal comando ripetitivo dovuta alla terminazione della sorgente per verificare se una ultima linea riempita con bianchi doveva essere stampata oppure no.
Nell'esempio 4.2.a, per ottenere lo stesso funzionamento, abbiamo dovuto inserire esplicitamente nel programma la trattazione della condizione di fine file di ingresso.

Tuttavia, dovrebbe risultare più semplice implementare efficientemente questa seconda proposta, proprio a causa della minore espressività delle sue strutture. Secondo Brinch Hansen i «Processi Sequenziali Comunicanti» sono più adatti ad una ricerca di tipo teorico sui problemi della concorrenza e a un uso come linguaggio di specifica, conciso e estremamente pulito, che come strumento di lavoro per un programmatore di sistemi.

Non ci sentiamo di concordare pienamente con questa affermazione, pur ritenendo che un lavoro di modifica, e probabilmente di estensione (con l'introduzione di un certo grado di ridondanza per rendere più controllabile il funzionamento dei programmi in fase di compilazione), deve essere fatto per renderlo effettivamente utilizzabile.

## 5. CONCLUSIONI

I due linguaggi che abbiamo preso in considerazione hanno senza dubbio alcune caratteristiche che li rendono estremamente attraenti: la possibilità di specificare in modo altamente modulare le funzionalità di un sistema, e di distribuirle in modo che ogni processo possa fare solo poche cose, ma efficientemente; l'eleganza con cui, soprattutto col linguaggio di Hoare, si risolvono i problemi della comunicazione tra processi (eleganza concettuale, non soltanto sintattica); l'estrema flessibilità e semplicità che essi hanno.

Tuttavia, se questi linguaggi rappresentano un primo passo verso lo sviluppo di sistemi altamente paralleli, e di strumenti linguistici appositamente tagliati per applicazioni su questi sistemi, molto lavoro deve essere ancora fatto nel campo dello sviluppo di architetture idonee e di metodi per la prova di correttezza.

La presenza di effetti laterali (che sono il principio di funzionamento dei «Distributed Processes» di Brinch Hansen), il nondeterminismo, la presenza di problemi come deadlock e starvation rendono estremamente difficile ottenere l'affidabilità che le applicazioni della programmazione concorrente richiedono.

In questo senso, una prospettiva di soluzione radicale, seppure ancora non ben definita, ci sembra essere quella offerta dall'uso di linguaggi e sistemi guidati dai dati. Alcuni lavori in questa direzione sono stati recentemente presentati [ 1,5 ].
La discussione su questi punti è ancora aperta, e sarà soggetto di ulteriore ricerca.

## 6. BIBLIOGRAFIA

[ 1 ] Arvind, Gostelow, K.P., Plouffe, W., An Asyncronous Programming Language and Computing Machine, Dept. of Information and Computer Science, Univ. of California, Irvine, Dic. 1978.

[ 2 ] Brinch Hansen, P., The Architecture of Concurrent Programs, Prentice-Hall, Englewood Cliffs, New Jersey, 1977.

[ 3 ] Brinch Hansen, P., Staunstrup, J., Specification and Implementation of Mutual Exclusion, IEEE Trans on Soft. Eng., SE-4,5, sett. 1978.

[ 4 ] Brinch Hansen, P., Distributed Processes: A Concurrent Programming Concept, Comm. ACM, 21,11, Nov. 1978.

[ 5 ] De Francesco, N., Perego, G., Tomasi, A., Vaglini, G., Vanneschi, M., On the Feasibility of Nondeterministic and Interprocess Communication Constructs in Data-Flow Computing Systems, 1st. European Conference on Parallel and Distributed Processing, Toulouse, Feb. 1979.

[ 6 ] Dijkstra, E.W., A Discipline of Programming, Prentice-Hall, Englewood Cliffs, 1976.

[ 7 ] Hoare, C.A.R., Monitors: An Operating Systems Structuring Concept, Comm. ACM, 17,10, Ott. 1974.

[ 8 ] Hoare, C.A.R., Communicating Sequential Processes, Comm. ACM, 21,8, Agosto 1978.

[ 9 ] Jammel, A.J., Stiegler, H.C., Structural Decomposition and Distributed Systems, 1st. European Conference on Parallel and Distributed Processing, Toulouse, Feb. 1979.

[ 10 ] Milne, G.J., Milner,R., Concurrent Processes and their Syntax, Draft, Dept. of Computer Science, Univ. of Edinburgh, Scotland, Aprile 1977.

[ 11 ] Milner, R., Examples and Proofs Concerning the Process Semantics of Infinite Nets, Draft, Dept. of Computer Science, Univ. of Edinburgh, Scotland, Giugno 1977.

[ 12 ] Price, R.J., Multiprocessing Made Easy, National Computer Conference, 1978.

[ 13 ] Wirth, N., Modula: a Language for Modular Multiprogramming, Software-Practice and Experience, 7,3-35, 1977.

# AVVENIMENTI

## CORSO SU "INFORMATION SYSTEMS, ORGANIZATIONAL CHOICE, SOCIAL VALUES", PISA, 9—20 APRILE 1979

### 1. Motivazione e obiettivi del corso

Il disegno di sistemi informativi basati sul computer è un'attività complessa. Tra l'altro, perchè deve tenere conto dei veloci cambiamenti della tecnologia (sia quella utilizzata dal sistema informativo stesso, sia quella su cui si applica l'organizzazione in esame), ed in generale dei mutamenti dell'ambiente in cui il sistema è inserito. Inoltre, deve tenere conto delle conseguenze organizzative e sociali determinate dall'introduzione o dalla modifica di un sistema informativo in strutture organizzative complesse quali sono le odierne imprese o la pubblica amministrazione.

Nonostante l'evidenza di queste considerazioni, si tratta di temi che non sono storicamente familiari al mondo dell'informatica e che solo negli ultimi anni si stanno imponendo per la loro rilevanza, anche a fronte del fallimento di molti progetti di informatizzazione, fallimento (parziale, cioè in termini di benefici ottenuti, o anche totale) imputabile alla sottovalutazione dei problemi indicati.

Anche in Italia, forse in modo più marcato che altrove, si riscontra su ciò una generale mancanza di attenzione dovuta al ritardo con cui le tecnologie ed i loro impatti raggiungono il nostro Paese, ma anche ad una carenza culturale comune al mondo della ricerca, al mondo imprenditoriale e al mondo sindacale.

Il corso, promosso dall'Information Training Group del Comitato per la Ricerca Scientifica e Tecnica (CREST—ITG) della Comunità Economica Europea, dal Consiglio Nazionale delle Ricerche e dall'Università di Pisa, si proponeva di rispondere ad un'esigenza reale: portare in Italia un dibattito, dei temi di ricerca, delle indicazioni concrete anche se non definitive, su cui un significativo insieme di studiosi sta lavorando in vari paesi.

È possibile individuare tre filoni in cui inquadrare i contributi presentati:

a) aspetti metodologici dello sviluppo dei sistemi informativi;
b) impatti sociali delle nuove tecnologie (microprocessori, Computer Assisted Design, Computer Assisted Instruction, ecc.);
c) partecipazione dei lavoratori e problemi sindacali connessi all'informatizzazione dell'industria e della Pubblica Amministrazione.

Una panoramica anche sintetica, ma non riduttiva, di tali contributi supera i limiti di un breve resoconto. Preferisco quindi centrare l'attenzione sull'approccio che si può definire socio-tecnico ai problemi metodologici, sviluppato e sperimentato da un gruppo di ricercatori di cui fanno parte Erich Mumford della Manchester Business School, Frank Land della London School of Economics e Jim Taylor del Center for Quality of Working Life dell'Università di California che lo hanno presentato a Pisa.
Nel secondo paragrafo sono presentate le motivazioni di base, nel terzo il "participative approach", nel quarto la metodologia di analisi proposta, nel quinto, infine, uno strumento specifico di supporto nella fase di analisi.

### 2. Perchè è necessario modificare l'abituale approccio al disegno dei sistemi informativi basati sul calcolatore

Numerosi fattori indicano la necessità di passare da un approccio al disegno dei sistemi informativi di tipo essenzialmente tecnologico ad uno basato sul concetto di sistema, in cui pertanto siano tenuti presenti oltre agli aspetti tecnologici anche quelli organizzativi ed, in generale, le esigenze dell'utente.

Esaminare questi fattori è utile per determinare le linee verso cui si deve indirizzare tale modifica. Land individua, tra gli altri:

1) la scarsa simpatia con cui il pubblico vede i sistemi basati sul calcolatore e la scarsa confidenza che la maggior parte delle persone ha verso i calcolatori, nonostante la loro sempre più vasta diffusione;

2) l'esistenza di una dinamica sociale verso uno spostamento di responsabilità e potere dall'alto al basso delle strutture organizzative. Ciò si traduce in movimenti sociali che premono per essere coinvolti nel determinare le decisioni che modificano l'organizzazione del lavoro attribuendo in esso sempre maggiore importanza a "valori umanistici" rispetto a fattori tecnologici [ 1 ]. In alcuni paesi [ 3 ] ciò si è tradotto in atti legislativi che obbligano le aziende a consultare i propri dipendenti prima di poter attuare un intervento che modifichi l'organizzazione del lavoro;

3) i frequenti fallimenti che si sono registrati nello sviluppo di sistemi EDP, dovuti a carenze di comunicazione tra utenti e specialisti EDP (gli utenti che non riescono a valutare le conseguenze degli interventi proposti, gli specialisti EDP che si costruiscono un modello del processo che è una lontana approssimazione della realtà);

4) l'evoluzione della tecnologia verso il disegno di sistemi distribuiti in cui uffici funzionalmente distinti dispongono del proprio calcolatore, che fa parte di un sistema più ampio. Ciò suggerisce di utilizzare metodi di disegno che consentano a ciascuno di sviluppare la propria parte.

Come è possibile modificare questa situazione? Per il gruppo di ricercatori citato la soluzione è nel "participative approach" [ 3 ]; si tratta cioè di sviluppare metodi che coinvolgano l'utente nel processo di disegno del sistema, in modo da tenere presente le esigenze di tutti i gruppi coinvolti e di diminuire le distanze tra utenti e specialisti EDP.

Ma se il ruolo dell'utente diventa centrale, è allora necessario precisare meglio chi è questo utente.

### 3. Chi è l'utente? I vari tipi di "Participative Approach"

All'interno di ogni organizzazione esistono in genere vari utenti di uno stesso sistema automatizato, e spesso gli interessi di alcuni tra questi entrano in conflitto.

Kistruck e Griffiths [ 4 ] identificano tre tipi di utenti in una organizzazione:

a) Gli individui o i gruppi che decidono la realizzazione di un nuovo sistema (o la modifica di uno preesistente) e che possono autorizzare lo sviluppo del progetto; sono i veri "clienti" del gruppo a cui è affidato il disegno del sistema ed in genere il loro approccio è di tipo strettamente economico (in termini di costi e profitti per l'azienda);usualmente hanno scarso contatto diretto con il sistema, pur ricevendone gli output.

b) I responsabili (quadri intermedi dell'azienda) delle funzioni in cui va inserito il sistema; sovente si sentono espropriati, da parte del sistema informativo automatizzato, di compiti di controllo che svolgevano in sua assenza e da cui traevano parte del loro potere; hanno quindi spesso un atteggiamento di difesa nei confronti degli esperti EDP ritenuti autori di tale espropriazione. È da questo gruppo di persone che talvolta prendono origine forme di organizzazione informale alternative a quella del sistema automatizzato, che lavorano in parallelo con essa.

c) Gli individui, o i gruppi, il cui lavoro è parte essenziale del sistema e senza il quale questo non potrebbe funzionare. Gli approcci convenzionali al disegno dei sistemi hanno posto scarsa attenzione alle esigenze ed aspirazioni di tale personale operativo, ma gran parte del successo del sistema risiede nella disponibiità e nell'interesse degli operatori ad utilizzare il sistema nel modo specificato.

Oltre a quelle indicate da Kistruck e Griffiths, Land individua due ulteriori classi di utenti:

d) Gli individui che si trovano al di fuori dell'organizzazione in cui è inserito il sistema automatizzato, ma che comunque hanno interazioni con esso (attraverso la bolletta della luce, l'estratto conto bancario, ecc...).
L'atteggiamento nei confronti dei calcolatori è in gran parte determinato da questo tipo di rapporti che ciascuno ha come cittadino al di fuori di una struttura organizzativa, ma poco è stato fatto perchè questa categoria di utenti sia presa in adeguata considerazione.
Mancano, ad esempio, specifiche associazioni di consumatori di sistemi automatizzati, e solo alcuni paesi hanno definito una legislazione che garantisca la riservatezza delle informazioni individuali.

e) Stefano Rodotà [ 5 ] ha sottolineato che, spesso, i più danneggiati dallo sviluppo di sistemi informativi sono particolari gruppi di individui

e che, quindi, la tutela dei diritti non si deve limitare a quella dei singoli, ma deve estendersi a quei gruppi che di volta in volta vengono toccati dalle scelte effettuate nel disegnare il sistema.

Land e Mumford propongono, sulla scorta di esperienze effettuate in ambito industriale [ 6 ], di coinvolgere l'utente in tutte le fasi di disegno del sistema, con differenti modalità in funzione del differente livello dell'utente all'interno dell'organizzazione. Si hanno così tre diversi livelli di partecipazione ("consultative", "representative", "consensus" participation), nati storicamente da una evoluzione dell'approccio generale, ma che si adattano rispettivamente agli utenti di livello a), b) e c).

Il "consultative approach" è il più appropriato per garantire un accordo su obiettivi di pianificazione di lungo periodo; il sistema è costruito dagli esperti EDP in stretto contatto con il top-management dell'azienda, ma con una ampia consultazione (principalmente attraverso interviste) di tutti i gruppi di livello gerarchico più basso toccati dal sistema automatizzato.

Il "representative approach" è ritenuto più appropriato per il disegno del sistema al livello del "middle management", cioè ove si tratti di coordinare differenti unità funzionali. Consiste nel costituire "gruppi di disegno" (*) in cui siano presenti (scelti o eletti) rappresentanti di tutti i livelli gerarchici dell'azienda coinvolti ed anche le organizzazioni sindacali.

Il "consensus approach" vuole consentire a tutto il personale di una unità coinvolta nel processo di informatizzazione di giocare un ruolo nel disegno del sistema. È utile quando debbano essere individuati con grande precisione (attraverso discussioni di piccoli gruppi) le esigenze di particolari unità dell'azienda.

## 4. L'approccio socio - tecnico all'analisi e al disegno del sistema

Benchè di fondamentale importanza per modificare il tradizionale approccio al disegno dei sistemi informativi basati sul computer, il "participative approach" non è tuttavia sufficiente a risolvere i problemi che usualmente si pongono. Land, Mumford e Taylor hanno perciò proposto alcuni strumenti ad hoc che possono rappresentare un utile supporto per gli esperti EDP.

L'approccio socio-tecnico [ 7, 8 ] all'analisi e al disegno del sistema propone di separare, in fase di analisi, gli aspetti tecnici da quelli sociali per ricomporli quindi in fase di disegno, risolvendo qui eventuali conflitti emersi dall'analisi.

La fase di analisi viene scomposta in cinque fasi:

**1. Descrizione della struttura organizzativa "essenziale" del sistema**, cioè di quelle funzioni che il sistema deve espletare per raggiungere i suoi obiettivi principali. Vanno individuati:

1.1. i confini del sistema (in termini di persone, tempo, territorio territorio, tecnologie) ed il suo ambiente (non tutto l'ambiente esterno, ma quelle parti di esso che interagiscono con il sistema in esame), prescindendo dall'organizzazione esistente;

1.2. la natura degli input del sistema (di primo livello, cioè le "cose" che devono essere essere trasformate);

1.3. la natura degli output del sistema (di primo livello, cioè i compiti fondamentali);

1.4. la natura del problema del management.

**2. Analisi tecnica**, che riguarda cioè le macchine, le procedure e le informazioni. Consiste di:

2.1. identificazione delle operazioni elementari, cioè di quell'insieme di attività integrate che realizzano una delle grandi funzioni del sistema e che sono separate dagli altri insiemi di attività da qualche tipo di confine;

2.2. descrizione dettagliata di ogni operazione elementare, prescindendo dalle attuali tecnologie procedure e strutture organizzative, in quanto queste potranno essere modificate in fase di disegno.

Per ogni operazione elementare devono essere individuate:

a) le attività necessarie per realizzare la sua funzione principale;

b) le varianze e gli strumenti per evitarle, o le attività per correggerle rapidamente quando si verificano (dove per varianza si intende la tendenza del sistema o di un sottosistema a deviare dal

(*) gli autori usano sovente un'accezione molto ampia del termine "disegno", comprendendo in essa anche la fase di analisi

comportamento standard).
A questo livello devono essere prese in esame solo quelle varianze che tendono a verificarsi a causa della natura degli obiettivi del sistema o delle funzioni essenziali di questo.

c) le attività di coordinamento all'interno di una operazione elementare o tra più operazioni elementari;

d) le attività di sviluppo (per incrementare l'efficienza e la adattabilità ai cambiamenti);

e) le attività di controllo richieste per aumentare l'efficienza delle singole operazioni elementari e del sistema totale.

**3. Analisi sociale**, cioè lo studio del sistema per quanto concerne gli aspetti umani e sociali. Consiste di:

3.1. specificazione dei ruoli e delle relazioni che devono essere associati alla struttura organizzativa "essenziale" (vedi 1) perchè il sistema possa realizzare i suoi obiettivi principali, uniti con le esigenze di 'job satisfaction' (soddisfazione nel lavoro) dei dipendenti che svolgono le mansioni collegate. A questo livello non devono essere assegnati dei compiti a singoli individui o gruppi, ma devono essere individuate le funzioni ed i livelli di conoscenza ed esperienza richiesti per ciascuna funzione. Solo in fase di disegno insiemi di funzioni verranno riuniti per formare insiemi di attività logicamente attribuibili a singoli individui o gruppi.

3.2. Le esigenze e le aspettative di 'job satisfaction' devono essere rese esplicite come parte della descrizione del sistema.
La 'job satisfaction' [ 9 ] è stata definita da E. Mumford come l'aderenza tra ciò che un individuo o un gruppo domanda alla sua situazione di lavoro e ciò che ne riceve.
Vari tipi di esigenze intervengono a determinare tale soddisfazione: esigenze della personalità, (di conoscenza, psicologiche), associate alla competenza ed all'efficienza nel posto di lavoro (mansioni, responsabilità, inquadramento, livelli salariali), esigenze di carattere etico (rapporti con gli altri dipendenti), ecc. In generale, il modo migliore per individuare queste esigenze è attraverso questionari, utilizzando poi i dati raccolti come base per discussioni in piccoli gruppi, finalizzate alla valutazione di come queste esigenze di 'job satisfaction' possano essere raggiunte.

**4. Analisi delle discrepanze.** Una volta descritto il sistema "essenziale" sia dal punto di vista tecnico che dal punto di vista sociale, è utile stabilire quanto la struttura organizzativa esistente faccia combaciare oppure metta in contrasto obiettivi ed esigenze di efficienza e di job satisfaction.
Questa analisi fornisce informazioni su quanto il sistema attuale diverge dal sistema richiesto.
L'analisi delle discrepanze identifica quelle varianze che sono frutto di procedure, organizzazione del lavoro, disponibilità delle risorse attualmente esistenti e serve come guida all'identificazione di quelle aree che non richiedono un ridisegno.

**5. Ridisegno del sistema.** Si tratta ora di produrre un nuovo disegno della struttura organizzativa del sistema che integri la parte tecnica e la parte sociale e consenta al sistema complessivo di raggiungere più efficientemente i suoi obiettivi principali. Un importante criterio di disegno è che il controllo delle varianze e il coordinamento necessario per raggiungere questo controllo sia posto ai livelli più bassi possibile in modo da poter essere maneggiato con successo.

Un esempio di applicazione dell'approccio sociotecnico all'analisi e al disegno di un sistema per un intervento EDP in tre divisioni di un grande laboratorio scientifico si trova in [ 10 ].

## 6. L'analisi del futuro

La decisione di realizzare cambiamenti in un sistema è generalmente basata su delle assunzioni sul minimo tempo che si prevede passare prima che il sistema sia soggetto ad un nuovo rilevante cambiamento.

Nel disegnare un nuovo sistema, o attuando cambiamenti su uno già esistente; si deve dunque sciogliere la contraddizione tra il progettare un sistema molto flessibile ma molto costoso oppure realizzare un sistema più economico ma anche meno adattabile.

Ciò che succede generalmente, in pratica, è invece il prolungamento della vita di un sistema non flessibile nel tentativo di prolungarne il tempo di vita.

Avere delle indicazioni sui cambiamenti a cui il sistema in realizzazione dovrà adattarsi nel corso del suo periodo di vita operativa può dunque essere utile per meglio valutare il tempo di ritorno degli investimenti, i possibili costi di manutenzione, le parti del sistema che richiedono maggiore flessibili-

tà e diminuire così le possibilità di un suo fallimento economico ed operativo.

Non si tratta ovviamente di predire il futuro, ma di individuare un metodo di analisi per dedurre dall'oggi le indicazioni sul domani. Land, Mumford e Hawgood [ 7 ] propongono quattro stadi di analisi:

1. Individuare i tipi di cambiamenti che possono avere un impatto sul sistema in esame, identificare un insieme di persone che possono aiutare a valutare le linee di sviluppo in tali ambiti, e scegliere degli strumenti utili per tentare delle previsioni.
L'individuazione dei possibili tipi di cambiamenti è facilitata dalla suddivisione proposta in cinque grandi categorie:

a) cambiamenti nella tecnologia disponibile (su cui si applica l'organizzazione, ad esempio la struttura del ciclo produttivo, oppure utilizzata dal sistema EDP stesso);
b) cambiamenti nel contesto giuridico;
c) cambiamenti nel contesto economico;
d) cambiamenti nelle attitudini, nelle attese, nel clima di opinioni dell'ambiente in cui il sistema è inserito;
e) cambiamenti interni all'organizzazione (nel management, nella struttura aziendale, ecc...).

Può essere utile per ognuno dei cambiamenti che possono avere impatti sul sistema individuare se sono qualitativi (ne alterano la 'logica'), o quantitativi (ne alterano il 'traffico', in più o in meno); se sono permanenti o temporanei.

2. Mediante discussioni tra il gruppo di disegno e gli esperti dei vari ambiti individuati in 1. e/o utilizzando i metodi (di modellizzazione, simulazione, statistici) individuati in 1., al secondo stadio è necessario precisare:

- quali cambiamenti dei tipi individuati in 1. si possono fin da ora prevedere e quando possono verificarsi;
- quali impatti possono avere sul sistema;
- quale probabilità hanno di verificarsi.

La risposta a queste domande fornisce l'output di questo secondo stadio, che consiste della definizione:

- del tempo di vita operativa del sistema che si vuole ottenere;
- della lista degli obiettivi del sistema, rivista alla luce dei possibili cambiamenti;
- della lista delle caratteristiche del sistema utili a garantirne la flessibilità.

3. Il gruppo di disegno deve verificare sistematicamente quali parti del sistema sono sensibili ad uno qualunque dei cambiamenti individuati in 2. È utile esaminare separatamente il sottosistema automatizzato ed il sottosistema manuale.
Per quanto concerne il sottosistema basato sul computer, il metodo proposto è il seguente:

a) listare tutti gli identificatori, e le loro relazioni: messaggi di input e di output, gli algoritmi e le norme procedurali, i files e/o le basi di dati;
b) individuare se è possibile far fronte ai cambiamenti in breve tempo e con limitato uso di risorse; in caso contrario valutarne le conseguenze e i costi di intervento a posteriori; l'analisi deve essere effettuata indipendentemente dalla probabilità di occorrenza dei vari cambiamenti.

Analogo studio deve essere effettuato per il sottosistema manuale, prendendo qui in considerazione anche gli aspetti sociali, individuali e/o di gruppo.

Il risultato di questo terzo stadio è la conoscenza dettagliata degli elementi del sistema proposto che sono più sensibili ai cambiamenti possibili e di quali, fra questi elementi, possono avere più conseguenze sull'intera organizzazione.

4. Il gruppo di disegno deve infine comparare, per tutti i punti critici individuati in 3., il costo derivante da un mancato adattamento del sistema rispetto al costo di costruire il sistema stesso in modo che risulti più flessibile.

### 6. Una linea di ricerca

Nonostante l'inevitabile sinteticità del resoconto, credo emerga la rilevanza dei temi affrontati per chiunque si trovi ad intervenire in una struttura organizzativa preesistente mediante automazione. È quindi indiscusso merito di questo gruppo di ricercatori l'avere da tempo intrapreso una difficile opera di sensibilizzazione del mondo dell'informatica sugli impatti sociali dei processi di automazione.

L'approccio che Land, Mumford e Taylor hanno presentato per affrontare questo insieme di problemi risente della formazione di carattere sociologico /organizzativo di molti di loro. I metodi proposti sono infatti ancora sensibilmente lontani da quelli dell'informatica: si può prospettare perciò il rischio di una scissione tra la fase di analisi in cui

vengono considerati anche gli aspetti organizzativi e sociali e la fase di vero e proprio disegno del sistema informativo, in cui le considerazioni di carattere tecnico finiscono col prevalere.
L'integrazione tra le due fasi e le due attitudini non può essere un problema affidato alla buona volontà dell'equipe, o del singolo, che disegna il sistema. Anche se è vero che esiste, e deve essere affrontato al più presto, un problema di formazione degli esperti EDP, la cui cultura di base non può più limitarsi agli aspetti tecnici [ 7 ],[ 11 ].

I significativi contributi di alcuni ricercatori italiani [ 12 ], e stranieri [ 10 ],[ 13 ], portatori di una attitudine più attenta alla integrazione sistematica degli aspetti tecnici con le problematiche sociali, ed in generale il dibattito che si è sviluppato durante il corso di Pisa, hanno mostrato che è necessario aprire una specifica area di ricerca, fortemente interdisciplinare, che abbia come oggetto l'analisi delle strutture organizzative, finalizzata al disegno di sistemi (informativi) basati sul calcolatore e che miri a definire una metodologia di analisi in cui alcune caratteristiche sono già individuabili:

- Un alto contenuto concettuale, in modo da poter descrivere esaustivamente, senza ambiguità e ridondanze, il sistema "essenziale", centrando cioè l'attenzione sulle sue funzionalità e prescindendo dalle strutture organizzative esistenti;
- la facilità di comprensione anche per i non specialisti, in modo da divenire un linguaggio di comunicazione fra esperti ed utenti e garantire così una partecipazione effettiva di questi all'analisi del sistema;
- la possibilità di integrazione con le usuali metodologie di disegno dei sistemi basati sul calcolatore, in modo che gli aspetti organizzativi e sociali siano tenuti presenti anche in fase di disegno.

F. De Cindio

**7. Bibliografia**

[ 1 ] T.D. Sterling, Guidelines for Humanizing Computerized Information Systems:a Report from Stanley House, Comm. ACM, vol. 17, no. 11, Novembre 1974

[ 2 ] Si veda ad esempio il "Data Agreement" tra Confindustria e Sindacati norvegesi, riportato in Uomini e Computer Come, gennaio 1979

[ 3 ] E. Mumford, F. Land, J. Hawgood, A Participative Approach to the Design of Computer Systems, Impact of Science on Society, vol. 28, no. 3, 1978

[ 4 ] J.R.S. Kistruck, A.R.B. Griffiths, Information Systems and their Social Construction, LSE Monograph Series on Information System Design, in pubblicazione nel 1979

[ 5 ] S. Rodotà, Privacy and Data Surveillance - A Growing Public Concern, in Policy Issues in Data Protection and Privacy, OECD Information Studies no. 10, 1976

[ 6 ] E. Mumford, D. Henshall, A Participative Approach to Computer Systems Design; A Case Study of the Introduction of a New Computer System, Associated Business Press, 1979

[ 7 ] J. Hawgood, F. Land, E. Mumford, Training the Systems Analyst of the 1980's Infotech State-of-the-Art Report, 1979

[ 8 ] J.C. Taylor, The Socio-Technical Approach to Work Design, in Designing Organizations for Satisfaction and Efficiency, a cura di K. Legge e E. Mumford, Gower Press

[ 9 ] E. Mumford, Job Satisfaction: A Method of Analysis, in Designing Organizations for Satisfaction and Efficiency, a cura di K.Legge e E. Mumford, Gower Press

[ 10 ] J.C. Taylor, Participative Socio-Technical Work System Analysis and Design: Service Technology Work Group, CQWL - WP - 78 - C, Center for Quality of Working Life, Institute of Industrial Relations, UCLA

[ 11 ] K. Nygaard, Tasks, Roles and Interests of Information System Specialists in the 1980's, lettura presentata al corso CREST, Pisa, aprile 1979

[ 12 ] A. De Maio, E. Bartezzaghi, O. Brivio, C. Ciborra, G. Zanarini, A New System Analysis Method Based on Socio-Technical System Approach, IFIP Conference, Oxford, aprile 1979

[ 13 ] J. Berleur, Some Methodological Problems to Build up a Regional Economic Data Base, lettura presentata al corso CREST, Pisa, aprile 1979

## 3° CONVEGNO SULLA PROGRAMMAZIONE STRUTTURATA, MILANO, 23-25 MAGGIO 1979

Sotto il patrocinio e con il coordinamento della Honeywell Information Systems Italia e in collaborazione con diverse Università italiane (Milano, Bologna, Torino, Pisa, Salerno, Politecnico di Milano) e con l'Istituto di Elaborazione dell'Informazione di Pisa del Consiglio Nazionale delle Ricerche, si è tenuto a Milano, presso l'Aerhotel Executive dal 23 al 25 maggio 1979, il 3° convegno sulla «Programmazione Strutturata».

Esso fa seguito ai convegni del 1974, quando la programmazione strutturata fu "lanciata" in Italia, sulla base delle prime esperienze fatte da alcuni gruppi pionieri e del 1976, che documentò il livello di esperienza raggiunto in Italia in differenti ambienti applicativi e industriali.

Il convegno ha visto la partecipazione di 270 specialisti e ricercatori, provenienti sia dai centri Universitari che da enti industriali, finanziari, commerciali e case di consulenza software.

I 18 contributi scientifici e tecnici presentati quest'anno (che si riferiscono a ricerche condotte presso i Centri Universitari che hanno organizzato con la Hisitalia il convegno o a esperienze realizzative effettuate presso industrie o in ambienti applicativi, finanziari o commerciali come l'ingegneria HISItalia, l'Olivetti, la SIT-Siemens, il Credito Italiano, la Telettra, lo CSELT) hanno messo in evidenza il processo, in corso, di transizione da una concezione iniziale della programmazione strutturata come tecnica per produrre programmi affidabili partendo da specifiche definite, a una visione più vasta e generale, dove le nuove metodologie sono applicate anche nelle fasi iniziali di analisi e di disegno.

Le esperienze degli ultimi anni di applicazione della programmazione strutturata hanno infatti portato a concludere che la chiave del successo di un progetto software (la costruzione di un prodotto affidabile, efficiente, facile da usare e rilasciato entro le date previste) sta nella cura, professionalità e completezza perseguite soprattutto nelle fasi iniziali del progetto (definizione dei "requirements" e delle specifiche funzionali). In effetti, in anni molto recenti, si è verificato uno spostamento di enfasi, nella ricerca metodologica nell'ambito della produzione di software, verso queste aree.

Questo aspetto è stato ben evidenziato da G. Torriani, direttore della Sezione Ingegneria di Software e di Sistema del Centro Progetti e Studi HISItalia di Pregnana Milanese, nel suo intervento di apertura dei lavori del convegno.

Questa estensione delle metodologie «strutturate» alle fasi di analisi e progettazione — ha affermato G. Torriani — si propone di produrre, prima della redazione vera e propria del codice dei programmi:

- specifiche funzionali complete, documentanti in modo esaustivo tutte le esigenze applicative dell'utilizzatore finale;
- una architettura del prodotto software che tenga conto di tutti i suoi possibili aspetti evolutivi;
- una definizione completa di tutti i passi del piano di realizzazione, in modo che si possa passare alle fasi di realizzazione e prova dei programmi:
  - . guidati da una pianificazione dettagliata;
  - . supportati da una valutazione realistica dei costi;
  - . capaci di fare previsioni accurate sulla qualità del prodotto finito.

Nel suo intervento, G. Torriani ha accennato anche all'impatto positivo che i metodi di analisi e di specifica produrranno sulla strada della definizione e produzione di componenti software standard, area di attività di cui si sta cominciando a capire l'importanza (vedi la rivista COMPUTER del febbraio 1979).

Il convegno ha trattato vari aspetti correlati con l'analisi e il disegno:

- concorrenza nei programmi e tecniche di «fault-tolerance» (3 lavori)
- organizzazione del progetto, management, formazione (2 lavori)
- metodologie di analisi e di progetto (5 lavori)
- linguaggi di alto livello e strumenti di produzione (7 lavori).

La HISItalia ha contribuito attraverso la presentazione di :

- PHOS (Program Hierarchy from Output Structure), una metodologia di progetto e programmazione per l'ambiente EDP, per la quale

la HISI tiene corsi regolari;

- Firmware Factory: la struttura e gli strumenti della fabbrica firmware in uso presso i laboratori di ricerca e sviluppo HISI di Pregnana Milanese.

La introduzione alle giornate è stata fatta da G. Degli Antoni (direttore dell'Istituto di Cibernetica della Università di Milano e condirettore del Communication and Programming Project, progetto di ricerca nelle metodologie software che la HISI conduce insieme con tale Istituto), che ha presentato le reti di Petri e il loro uso nella descrizione di procedure informatiche.

A. Cicu

Lavori presentati: V. De Antonellis, G. Degli Antoni, G. Mauri, B. Zonta, Una notazione per la descrizione delle procedure e del loro comportamento basata sulle reti di Petri; LA CONCORRENZA DEI PROGRAMMI E TECNICHE DI FAULT-TOLERANCE: P. Ancillotti, N. Lijtmaer, M. Boari, Meccanismi di allocazione e protezione in linguaggi per la programmazione concorrente; P. Ancillotti, N. Lijtmaer, M. Boari, A. Natali, Tecniche di Fault-Tolerance in sistemi concorrenti; M. Morganti, Una esperienza nel campo del Fault-Tolerance nell'ambito di un progetto per il controllo di una centrale di commutazione elettronica; ASPETTI ORGANIZZATIVI E MANAGERIALI: C. Barassi, Esperienza educativa e manageriale di introduzione della programmazione strutturata; G. Rosci, L'organizzazione dei progetti software nel Proteo; METODI DI SPECIFICA E DI PROGETTO: F. De Cindio, G. De Michelis, C. Simone, Una proposta per la definizione dei requisiti di un sistema EDP che utilizza i Guarded Commands; G. Castelli, G. Haus, M. Maiocchi, Una metodologia di stesura e verifica di specifiche funzionali di procedure e programmi software; A. Maserati, F. Monzani, E. Spoletini, E. Toscani, PHOS: una metodologia di programmazione per l'ambiente EDP; A. Di Leva, L. Petrone, Sviluppo strutturato dell'architettura di un sistema di gestione di basi di dati su minicalcolatori; C. Batini, Progetto Top-Down di basi di dati assistito da calcolatore, un'applicazione ai sistemi informativi socio-sanitari; LINGUAGGI E STRUMENTI: A. Camici, R. Martucci, Problematiche relative all'uso di linguaggi di descrizione dei sistemi e linguaggi di programmazione ad alto livello nel campo delle telecomunicazioni; P. Asirelli, A. Martelli, U. Montanari, Costrutti e linguaggi di programmazione ad alto livello nel campo delle telecomunicazioni; P. Asirelli, A. Martelli, U. Montanari, Costrutti e linguaggi di programmazione con controllo degli effetti laterali; C. Ghezzi, F. Tisato, Meccanismi elementari per la strutturazione del software; A. Caccamese, G. Malatesta, A. Osnaghi, T. Pedrotti, F. Tisato, Due strumenti integrati per la configurazione di sistemi software; A. Albani, C. Monducci, S. Dalla, M. Maiocchi, Una firmware factory per lo sviluppo strutturato di firmware di sistemi multiprocessor; P. Degano, G. Pacini, F. Turini, G. Levi, P. Sirovich, An Integrated System to Support Program Design, Development and Analysis; A. Celentano, P. Della Vigna, C. Ghezzi, D. Mandrioli, Un sistema integrato per lo sviluppo dei programmi.

## LA TERZA CONFERENZA INTERNAZIONALE HONEYWELL SULL'INGEGNERIA DI SOFTWARE, BLOOMINGTON, 27-29 MARZO 1979

Si è tenuta a Bloomington (Minnesota - Usa), dal 27 al 29 marzo 1979, la terza conferenza internazionale Honeywell sull'Ingegneria di Software.

Anche questa conferenza fa parte di un ciclo di incontri tra gli specialisti e i dirigenti dei Centri di Ingegneria di Software della Honeywell promossi dalla direzione della società, al fine di facilitare lo scambio di esperienze e il trasferimento di competenze fra gruppi diversi nel settore del software.

Il Corporate Computer Science Center, che ha il compito di condurre e promuovere attività di ricerca e sperimentazione avanzate nei vari settori in sviluppo della computer science sia per la Honeywell Information Systems che per la Honeywell Control Systems, e che nel settore dell'ingegneria del software sta sperimentando direttamente il metodo di progetto Wellmade (vedi Note di Software n. 5), supportandone anche la diffusione all'interno della società, ha individuato in queste conferenze uno dei modi più efficaci per un rapido aggiornamento reciproco dei gruppi software Honeywell sullo stato dell'arte interno ed esterno. Infatti, tutti i gruppi software Honeywell sono sollecitati a presentare in queste conferenze contributi qualificati derivanti dalle loro esperienze.

Le analoghe conferenze organizzate nel 1977 e nel 1978 avevano trattato rispettivamente la produttività nella produzione di software e le tecniche di management dei progetti software (vedi Note di Software n. 2 e 5).

Nella conferenza di quest'anno, si richiedeva ai partecipanti di porre enfasi sugli aspetti ritenuti più importanti e decisivi per la progettazione di un software affidabile, a costi predeterminati e alle date previste, e in particolare si richiedeva di trattare metodi di definizione e di controllo delle fasi iniziali di un progetto (la definizione dei "requirements" - gli obiettivi - e la definizione delle specifiche funzionali).

La conferenza ha visto la partecipazione di 140 capi e specialisti, rappresentanti vari settori della Honeywell, con 11 contributi provenienti dal settore della Honeywell Information Systems (HIS) e 14 provenienti dalla Honeywell Control Systems (HCS), a testimonianza dell'importanza crescente che il software - in particolare per i microprocessori - sta acquisendo non solo nei calcolatori "general purpose" (HIS) ma anche nelle applicazioni speciali dei sistemi di controllo (HCS).

La relazione introduttiva è stata tenuta da Irma Wyman, direttrice del gruppo centrale di staff dell'U.S Information Systems Group (la divisione americana della HIS). L'autrice, dopo aver affermato che il software diventerà entro pochi anni uno dei più importanti prodotti industriali, portando a sostegno di questa tesi una serie di dati sulle spese future degli utenti nel campo del software, ha esposto una stimolante sintesi delle direzioni in cui gli ingegneri software devono orientare i loro sforzi per migliorare tale prodotto:

- riempire la lacuna attuale degli standards software,poveri nell'indicare come controllare le attività e i prodotti

- definire criteri e metri di misura di prestazioni, di qualità e di produttività

- scegliere e applicare criteri di testing e di rilascio (nello hardware - ha affermato la Wyman - non si accetterebbe mai di consegnare un prototipo)

- elevare la qualità della documentazione soprattutto nel senso di renderne facile la lettura e l'apprendimento (la documentazione «è» il software per l'utente, il codice è incidentale: l'utente legge la documentazione, non il codice)

- manutenere il prodotto e supportare con efficienza il suo uso lungo tutto l'arco di vita

- soprattutto, divenire capaci di definire specifiche complete, non ambigue, non mutevoli.

Infine, Wyman ha evidenziato, attraverso una serie di considerazioni sistemistiche e tecnologiche, come le due maggiori tendenze in atto nell'informatica, e cioè

- il crescente diffondersi dei sistemi distribuiti

- l'uso dei sistemi reso sempre più facile allo end-user, all'utente non esperto,

renderanno il software un prodotto sempre più complesso con la conseguente ancor maggiore urgenza di investimenti nelle direzioni dette.

L'importanza dello "end-user" nel concepire e progettare i futuri prodotti software è stata in modo indiretto ma autorevole sottolineata nella "dinner conference" del convegno, per la quale è stato invitato uno dei più noti scienziati-futurologhi americani, E.C. Joseph.

Joseph ha svolto una stimolante e vasta esposizione di come la tecnologia VLSI (Very Large Scale Integration) e il relativo software per microprocessori riusciranno, grazie alle loro piccole dimensioni e ai loro costi ridotti, a entrare in tutti i settori della vita sociale e nella vita di ogni uomo,

- rendendo "intelligenti" le macchine che siamo abituati ad usare

- dando la possibilità di costruire nuove macchine (Joseph prevede per la fine degli anni 80 addirittura "scuole" e "uffici" portatili, e che le fabbriche diventeranno "macchine orientate alla comunicazione")

- dando a tutti (anche al singolo privato) delle possibilità impensate di comunicare, attraverso le reti di comunicazione, che si saranno oltremodo diffuse, e quindi di svolgere una gran parte di meetings e di lavoro senza viaggi e senza spostamenti fisici.

L'evoluzione verso macchine tutte "intelligenti" (che avverrà in pochi anni - Joseph ha presentato diversi esempi già esistenti) e verso una società basata sulle comunicazioni dei dati porterà ad una autentica rivoluzione:

- i beni prodotti saranno infinitamente più durevoli e meno inquinanti (graduale eliminazione della società dello spreco)

- il singolo uomo avrà più tempo per una continua

autoeducazione (e la diffusione di cultura costerà meno)

- la salute sarà curabile in un modo estremamente decentralizzato (stimolantissima l'esposizione dei devices miniaturizzati, computerizzati e indossabili che saranno disponibili a supporto del controllo e della cura della salute)

- le macchine, attraverso l'immissione in esse della appropriata "intelligenza" (immenso lavoro - e responsabilità - per i "softwaristi") diventeranno vere "amiche" dell'uomo e sarà "bello" usarle.

Questa immissione di "umanesimo" nel convegno è stata molto apprezzata dai partecipanti, perchè ha suggerito verso quale direzione essenziale dovranno essere ricercati e definiti i "requirements" dei futuri sistemi e delle future applicazioni (e cioè il rendere più facile, più piacevole e più economico il lavoro all'uomo) e quale immenso spazio ci sarà per una creatività costruttiva.

I 25 contributi tecnici precedentemente citati hanno visto la seguente distribuzione per argomento:

| | |
|---|---|
| - definizione e validazione delle esigenze (requirements) | 5 contributi |
| - coinvolgimento dell'utente finale nella valutazione e validazione dei requirements | 4 |
| - le fasi del processo di sviluppo | 3 |
| - tecniche di testing (inclusa la verifica dei requirements) come parte integrante di ogni fase di sviluppo | 9 (1 HISItalia) |
| - previsone dei costi di progetto | 1 |
| - definizione delle specifiche funzionali | 2 (entrambi HISItalia) |
| - misura delle prestazioni | 2 (1 HISItalia) |
| - verifiche formali di correttezza | 1 |
| - uso del metodo di progetto WELLMADE | 2 |

I primi tre contributi HISItalia citati qui sopra sono un risultato dell'attività del Communication and Programming Project in collaborazione tra HISItalia e Istituto di Cibernetica dell'Università di Milano.

Tutti i lavori avrebbero ampi motivi di essere singolarmente riassunti, ma per essi si rimanda agli atti del convegno.

Si ritiene, tuttavia, opportuno evidenziare il lavoro di Jack Ring, del Large Information Systems Department della HIS di Phoenix, dal titolo "Achieving an Adequate, Accurate and Timely Set of Software Requirements" per l'analisi approfondita che vi viene esposta del processo di definizione dei requirements, e i due lavori italiani sui metodi di definizione delle specifiche funzionali di procedure EDP, uno basato sull'uso delle reti di Petri, l'altro sull'uso dei "guarded commands", che hanno suscitato notevole interesse; come pure notevole interesse ha riscontrato il contributo italiano sulla misura delle prestazioni, specialmente per la sua aderenza al profilo applicativo dell'utente.

A. Cicu

Indice dei lavori: J. Ring, Achieving an Adequate, Accurate and Timely Set of Software Requirements; R.S. Parker, A Review of the Space Shuttle Flight Control System Software Development; G.C. Loper, Use of Field Trials in New Product Definition; H. Van Calcar, D. Peterson, Oil Platform Installation Trainer Software Engineering; H.A. Bjorklund, Price Software Model; F. De Cindio, G. De Michelis, C. Simone, A Discipline for the Analysis of EDP Systems Using the Guarded Commands; G. Haus, G. Castelli, M. Maiocchi, A Methodology for the Construction and the Verification of Functional Specifications for Software Procedures and Programs; E. L. Averill, Requirements Methodology; A. Holdridge, An Additional Use for a Software Event Trace; I.L. Vivaldi, Performance Evaluation of Level 6 Software Systems; R.D. Stinaff, D.L. McNally, An Evaluation and Comparison of Two Software Design Methodologies; J.M. Kamrad II, Approach Towards Testing During the Design Stage; E.R. Rang, Verification Methodologies for Flight Control Systems; G. Degli Antoni, Petri Nets and Software Development Problems; W.E. Boebert, J.M. Kamrad, Proof of Security and Formal Specifications; W.L. Persons, On Software Myth Number 2 - Testing as a Means of Improving Software Reliability; R.R. Rakowski, Practical Guidelines for Software Testing - A

Communication Tester' Viewpoint; S. Rossini, HISI Approach to Performance Evaluation and Competition Analysis; J. L. Carr, Using Our Computer Resources as a Tool in Maintaining Software Quality; P.R. Rebelein, Design Verification Testing for a "Man Rated" Software System; P. Pfaffman, Commonalities and Differences in Systems and Software Testing; M. Bancroft, Managing the Small Automation Project - The System Engineer' Challenge; M.L. Murdock, MASCOT, A Modular Approach to Software Construction, Operation and Test; A. Cicu, M. Ceriani, M. Maiocchi, R. Polillo, A Method (and Related Tools) Devoted to Accurate Test Specification and Design and Test Auditing Activities; G.H. Colliat, Application of WELLMADE to a Large Scale System Development; S.E. Minear, An Application of WELLMADE to a Small One Man Project; J.D. Baker, Closing Remarks.

# IL SEGNALIBRO

In questa rubrica vengono segnalati libri, articoli o studi di recente pubblicazione che, a giudizio della redazione o dei lettori di NOTE DI SOFTWARE, rivestono un particolare interesse nell'ambito dei temi a cui questa rivista è dedicata.
Le citazioni fra virgolette, se non ne è indicata la fonte, sono tratte dai lavori stessi.

**Un libro sulla 'Performance Evaluation'**

● D. Ferrari, G. Serazzi, A. Zeigner, LE PRESTAZIONI DEGLI ELABORATORI ELETTRONICI - MISURA / VALUTAZIONE / OTTIMIZZAZIONE, Franco Angeli Editore, Collana dei Quaderni di Informatica, pagg. 535, 1979, Lit. 16.000

"Uno degli aspetti di maggior rilievo nei più recenti sviluppi dell'informatica è il crescente approfondimento degli studi sulle prestazioni dei sistemi informatici alla cui straordinaria crescita si è via via affiancata la diffusione sempre più ampia di tecniche, metodi e strumenti atti a indagarne gli aspetti particolari. Rispetto ad altri sull'argomento, questo volume presenta due caratteristiche salienti: nella sua impostazione di fondo persegue l'obiettivo di dare al lettore tutti gli elementi fondamentali riguardanti i metodi e gli strumenti per operare con efficacia nel campo della valutazione dei sistemi informatici, mentre nello sviluppo della materia dà largo spazio agli aspetti pratici della misurazione delle attività dell'hardware e del software degli impianti di elaborazione dei dati. Esso colma inoltre la sentita lacuna della mancanza di testi in italiano sull'argomento, sviluppando in maniera organica quanto è possibile trovare sparso in numerose pubblicazioni estere.
La struttura del volume è studiata in modo da rendere evidenti due livelli di lettura: quello informativo generale (paragrafi), in cui vengono affrontati gli argomenti con l'aiuto di esempi pratici, e quello più approfondito (sottoparagrafi), in cui sono descritte anche alcune implicazioni teoriche. Alla descrizione di "cosa" misurare, e cioè degli indici di prestazione più interessanti, segue quella di "come"misurare, e cioè delle varie tecniche possibili. Particolare attenzione è dedicata alla caratterizzazione del carico mediante la realizzazione dei modelli ed alle varie rappresentazioni delle grandezze rilevate. La rassegna dei più tipici strumenti di misurazione è integrata da una presentazione delle metodologie che devono essere adottate quando si pianifica e si affronta una sessione di misurazione. Gli strumenti e le metodologie descritte vengono quindi applicate all'ottimizzazione di alcuni sistemi, al fine di fornire un'utile traccia a quanti debbono effettuare studi di valutazione. Il problema dell'ottimizzazione del carico è affrontato con l'analisi delle possibili tecniche di ottimizzazione dei programmi che permettono un miglioramento del codice e dell'attività di I/O e una riduzione della frequenza di paginazione. Vengono inoltre illustrate alcune applicazioni di strumenti software commerciali.
Poichè la valutazione economica è una parte indispensabile in ogni studio, ad essa viene dedicato un capitolo nel quale più che alla descrizione delle possibili tecniche, si è data particolare importanza all'analisi dei costi e dei benefici relativi ad alcuni casi pratici di ottimizzazione dei sistemi e del carico.
Un'introduzione alla "fisica del software" ed un glossario concludono il libro che, con oltre 174 figure e 58 tavole, può essere considerato uno strumento formativo e di consultazione particolarmente utile sia a chi vuole accostarsi all'argomento sia a chi, già esperto, desidera integrare le proprie conoscenze." (copertina)

Indice: 1. Definizione del problema: 1.1. Definizioni e concetti di base; 1.2. Gli obiettivi della valutazione; 1.3. I sistemi di riferimento; 1.4. Gli indici di prestazione; 1.5. Tecniche di valutazione. 2. Tecniche di misurazione: 2.1. Generalità; 2.2. Misurazioni per il rilevamento di eventi; 2.3. Misurazioni con carichi pilota (benchmarking); 2.4. Misurazioni per campionamento; 2.5. Simulazione. 3. Rappresentazione delle grandezze misurate: 3.1. Introduzione; 3.2. Tabelle e grafici. 4. Strumenti di misurazione: 4.1. Le caratteristiche principali di uno strumento; 4.2. Strumenti software (software monitors); 4.3. Strumenti hardware (hardware monitors); 4.4. Simulatori. 5. Metodologie di valutazione: 5.1. La definizione degli obiettivi; 5.2. La scelta dello strumento; 5.3. La pianificazione di una sessione di misurazione; 5.4. Ricerca delle strozzature (bottlenecks); 5.5. L'eliminazione delle strozzature. 6. Ottimizzazione dei sistemi: 6.1. Generalità; 6.2. Il bilanciamento di un sistema multiprogrammato (SR/BM); 6.3. L'ottimizzazione di un sistema interattivo (SR/IM); 6.4. Il miglioramento di un sistema con memoria virtuale;

6.5. Il controllo delle prestazioni di un sistema per la gestione di banche di dati; 6.6. Bilanciamento ottimale del carico su unità di I/O con un modello analitico. 7. Ottimizzazione del carico: 7.1. Criteri generali e scelta dei programmi da ottimizzare; 7.2. Tipi di ottimizzazione dei programmi; 7.3. Riduzione del tempo di esecuzione. 8. Considerazioni economiche: 8.1. Generalità; 8.2. Costi e ricavi di uno studio di valutazione. Appendice A: Nozioni di fisica del software (K.W. Kolence). Appendice B: Glossario.

**Linguaggi di programmazione**

● Shaw, M., Hilfinger, P., Wulf, W.A., TARTAN — LANGUAGE DESIGN FOR THE IRONMAN REQUIREMENT: REFERENCE MANUAL, SIGPLAN Notices (ACM), vol. 13,N. 9, settembre 1978, 36-58

"Tartan is an experiment in language design. The goal was to determine whether a "simple" language could meet substantially all of the ironman requirement for a common high-order programming language.
We undertook this experiment because we believed that all the design done in the first phase of the DOD effort were too large and too complex. We saw that complexity as a serious failure of the designs: excess complexity in a programming language can interfere with its use, even to the extent that any beneficial properties are of little conseguence. We wanted to find out whether the requirements inherently lead to such complexity or whether a substantially simpler language would suffice.

Three ground rules drove the experiment. First, no more than two months — April 1 to May 31 — would be devoted to the project. Second, the language would meet all the ironman requirements except for a few points at which it would anticipate Steelman requirements. Further, the language would contain no extra features unless they resulted in a simpler language. Third, simplicity would be the overriding objective.
The resulting language, Tartan, is based on all available information, including the designs already produced. The language definition is presented here: a companion report provides an overview of the language, a number of examples, and more expository explanations of some of the language features.
We believe that Tartan is a substantial improvement over the earlier designs, particularly in its simplicity. There is, of course, no objective measure of simplicity, but the syntax, the size of the definition, and the number of concepts required are all smaller in Tartan.
Moreover, Tartan substantially meets all of the ironman requirements (the exceptions lie in a few places where we anticipated Steelman requirements and where details are still missing from this report).Thus, we believe that a simple language can meet the ironman requirement. Tartan is an existence proof of that.
We must emphasize again that this effort is an experiment, not an attempt to compete with DOD contractors. Tartan is, however, an open challenge to the Phase II contractors: the language can be at least this simple! Can you do better?"

● Shaw, M., Hilfinger, P., Wulf, W.A., TARTAN — LANGUAGE DESIGN FOR THE IRONMAN REQUIREMENT: NOTES AND EXAMPLES, ibidem, 59-75

"[ ... ] This report provides a more expository discussion of some of the language features, some examples of its use, and a discussion of some facilities that could enhance the basic design at relatively little cost."

● Il numero di ottobre 1978 di SIGPLAN Notices (ACM) (vol. 13, n. 10), pagg. 10-32 contiene quattro lettere di E.W.Dijkstra, a commento dei linguaggi di programmazione BLUE, GREEN, YELLOW, RED, candidati per il 'DOD common programming language effort'.

**Il nuovo standard Fortran**

● Brainerd, W., editor, FORTRAN 77, Communications of the ACM, vol. 21, n. 10, ottobre 1978, 806-820

"There is a new standard Fortran. The official title is "American National Standard Programming Language Fortran, X3.9-1978," but it is more commonly referred to as "Fortran 77," since its development was completed in 1977. It replaces the Fortran standard designated X3.9-1966. This paper describes many of the features of Fortran 77 and also provides some information about how and why the standard was developed."

**Multiprogrammazione**

• Brinch Hansen, P., DISTRIBUTED PROCESSES: A CONCURRENT PROGRAMMING CONCEPT, Communications of the ACM, vol. 21, n. 11, novembre 1978, 934-941

"A language concept for concurrent processes without common variables is introduced. These processes communicate and synchronize by means of procedure calls and guarded regions. This concept is proposed for real-time applications controlled by microcomputer networks with distributed storage. The paper gives several examples of distributed processes and shows that they include procedures, coroutines, classes, monitors, processes, semaphores, buffers, path expressions, and input/output as special cases."

**Flow-Charts**

• FULL REPORT OF THE FLOWCHART COMMITTEE ON ANS STANDARD X3.5—1970, SIGPLAN Notices (ACM), vol. 14, n. 3, marzo 1979, 16-27

"This report covers the recommendations of the Flowchart Committee, a brief history of the Committee's work, and the membership of the Committee.[ ... ] The Flowchart Committee recommends that a full comparative study be made of the value and contribution in computer program and system development and maintenance of all forms of flowcharts and alternatives to flowcharts. An ANS Flowchart Standard has existed for about 15 years. Yet no authority in the computer field advocates the ANS Flowchart Standard as the preferred technique for use in the development or maintenance of computer software. Instead, authorities decry it. No computer user in the world outside of the U.S. Government advocates and uses the ANS Standard flowchart as either the preferred technique or as a required or optional supporting technique. Even in the U.S. Government, the advocacy is minimal, and the actual practice is rare. No pubblication serving the computer field either from an AFIPS constituent society or a commercial publisher requires flowcharts it publishes to conform to the ANS Standard. Indeed, even people and organizations which know about the ANS Flowchart Standard have nearly always rejected it in favor of other techniques, especially during the past half-decade. Faced with this reality, the Flowchart Committee believes that to advocate a minor or cosmetic revision of the X3.5-1970 ANS Standard would be worse than pointless. It would tend to give an official blessing to a technique that may no longer be generally viable. Yet a careful independent comparative examination of the alternatives claimed to be superior has never been done. [ ... ]"

**La serie di informatica della Isedi**

È uscito il numero 8 della Serie di Informatica della Collana "Testi Scientifici Modulari" della ISEDI (vedi NOTE DI SOFTWARE 3 per i volumi 1 e 2, e NOTE DI SOFTWARE 6/7 per i volumi 3,4,5,6 e 7).

• Bracchi, G., Martella, G., Pelagatti, G., SISTEMI PER LA GESTIONE DI BASI DEI DATI, Collana TSM, Serie di Informatica, 8, ISEDI, Milano (1979), pagg. xvi + 220, Lire 7.000

"Questo volume costituisce un'introduzione avanzata alle tecniche dei Sistemi per la gestione di Basi dei Dati (o brevemente DBMS). Dapprima si approfondisce gradualmente il concetto di DBMS come strumento generalizzato per la gestione di raccolte integrate di archivi, basato su un software orientato all'interrogazione, all'aggiornamento, alla manutenzione e alla protezione della qualità, della privatezza e integrità delle informazioni registrate su memorie di massa. Segue una trattazione analitica dei fondamenti teorici e degli aspetti tecnico funzionali dei DBMS, con particolare riferimento ai modelli dei dati, ai linguaggi di interrogazione e manipolazione, all'architettura e alle soluzioni dei vari sistemi esistenti.

I problemi collegati ai DBMS e relativi all'organizzazione fisica degli archivi sulle memorie di massa vengono trattati a parte nel volume Tecniche di organizzazione degli archivi degli stessi Autori pubblicato in questa serie di Informatica." (copertina)

Indice: 1. Concetti fondamentali: 1.1. La Base dei dati; 1.2. Architettura di un DBMS; 1.3. Modo di operare di un DBMS; 1.4. Un esempio di DBMS completo; 1.5. Classificazione di alcuni DBMS esistenti. 2. Modelli dei dati e linguaggi: 2.1. I modelli gerarchico, relazionale e reticolare; 2.2. I linguaggi DML; 2.3. Commenti conclusivi. 3. L'IMS: 3.1. Terminologia IMS; 3.2. Strutture fisiche dell'IMS; 3.3. Estensione al modello dei dati; 3.4. Linguaggi non procedurali per i sistemi gerarchici. 4. La proposta CODASYL: 4.1. Caratteristiche generali della proposta; 4.2. La definizione dei dati in DDL; 4.3. La manipolazione dei dati in DML; 4.4. Un esempio; 4.5. La strategia di posizionamento dei record nel DMS 1100; 4.6. Commenti conclusivi.

5. Il modello relazionale: 5.1. La normalizzazione delle relazioni; 5.2. Il calcolo relazionale; 5.3. I sottoschemi in un sistema relazionale; 5.4. Altri linguaggi per il modello relazionale; 5.5. Considerazioni sull'implementazione. 6. Efficienza del sistema: 6.1. Efficienza e indipendenza dei dati; 6.2. Valutazione dell'efficienza; 6.3. Ottimizzazione dell'efficienza; 6.4. Riorganizzazione della base dei dati. 7. Indipendenza dei dati: 7.1. Introduzione; 7.2. Realizzazione dell'indipendenza dei dati. 8. La sicurezza dei dati: 8.1. La privatezza dei dati; 8.2. Integrità; 8.3. Uso concorrente; 8.4. La sicurezza nell'IMS e nel sistema CODASYL DBTG. 9. Tendenza dei DBMS: 9.1. Architettura del DBMS e modelli dei dati; 9.2. DBMS e valutazione delle prestazioni; 9.3. DBMS e reti di calcolatori; 9.4. DBMS e architettura dei sistemi di elaborazione; 9.5. DBMS, teoria dei linguaggi e ingegneria del software; 9.6. DBMS e intelligenza artificiale; 9.7. DBMS e tecniche di analisi dei sistemi informativi; 9.8. Commenti conclusivi.

**Project Management**

● Lehman, J.H., HOW SOFTWARE PROJECTS ARE REALLY MANAGED, DATAMATION, vol. 25, n. 1, gennaio 1979, 119-129

"Software engineering was introduced in the 1970s to apply the stronger disciplines of engineering (in contrast to the "art"of computer programming) to the design of computer software. This was done in an attempt to solve the problem of, or at least reduce the severity of, software development failures.
Similar advances have not been heralded in the area of software engineering project management, however. Does this mean that the old procedures still apply? Or have management techniques somehow kept pace? Just what are the trends in managing software development today?
To get some kind of answer to these questions, we approached the members of the American Inst. of Aeronautics and Astronautics (AIAA) Technical Committee on Computer Systems. We asked if they, as representatives of major firms in the aerospace industry, would be willing to participate in a survey. They were interested, and what follows is a compilation of how those aerospace firms manage software projects [ ... ]"

● Crossman, T.D., TAKING THE MEASURE OF PROGRAMMER PRODUCTIVITY, DATAMATION, vol. 25, n. 5, maggio 1979, 144-147

"[ ... ] Programmer productivity is a dilemma. On the one hand, we want to control projects by knowing exactly when and where slippages occur, we want to know if our programmers are working efficiently, we want to know if using a new methodology really is beneficial. We despise estimates that are based on "gut feel", but it appears that if we are to measure the productivity of our programmers we have to identify project variables and calculate their influence on our programming staff, make subjective assessments of the envisaged complexity of programs and the predicted ability of programmers, clarify terminology that has no industry-accepted definitions, measure the quality of our programmers' work, and base programmer performance on project estimates (which are arrived at unscientifically, anyway). It may be easier to say it just cannot be done.[ ... ] We explored this idea of measuring productivity in terms of program functions. We took statistics from a selection of six systems developed in our department during 1977. For each system, we divided the number of manhours spent on system development by the number of functions in all the programs in the system.[ ... ] The ratios resulting form this division of development time by functions were either very close to 2 or very close to 4. We found that the common factor in those systems with a ratio very close to 4 was that they were all developed using either a new language, a new operating system, new transaction processor software, or new data base software [ ... ]"

**Varie**

● Cale, E.G., Gremillion, L.L., McKenney, J.L., PRICE/PERFORMANCE PATTERNS OF U.S COMPUTER SYSTEMS, Communications of the ACM, vol. 22, n.4, aprile 1979, 225-233

"Econometric models of the U.S. computer market have been developed to study the relationships between system price and hardware performance. Single measures of price/performance such as "Grosch's Law" are shown to be so oversimplified as to be meaningless. Multiple-regression models predicting system cost as a function of several hardware characteristics do, however, reveal a market dichotomy. On one hand there exists a stable, price predictable market for larger, general purpose computer systems. The other market is the developing one for small business computer systems, a market which is relatively unstable with low price predictability."

**Honeywell**
Honeywell Information Systems Italia
Via Laboratori Olivetti - Pregnana Milanese

**UNIVERSITA'**
**DEGLI STUDI DI MILANO**
ISTITUTO DI CIBERNETICA
Via Viotti, 5 - Milano

**Honeywell**
Honeywell Information Systems Italia

NOTESW9 00052T 01

CIMINO MICHELE

HONEYWELL I.S.I.

VIA PIRELLI 32
20124MILANO

Ufficio Documentazione Gestione Mailing List
Via Tazzoli 6-**20154 MILANO**